MASTER EN SCIENCES INFORMATIQUES

Implémentation d'un protocole de sécurité pour la carte à puce générique

Bianco, Marc

Award date:
1997

Facultés Universitaires Notre-Dame de la Paix, Namur
Institut d'informatique
Année académique 1996-1997

Implémentation d'un protocole de sécurité pour la carte à puce générique

Marc Bianco

Mémoire présenté en vue de l'obtention du grade de Licencié et Maître en informatique

Remerciements

Au cours de ce stage, bon nombre de personnes sont intervenues afin que mon travail se réalise dans les meilleures conditions. C'est pourquoi je voudrais remercier les membres de l'équipe Recherche et Développement de Gemplus : Pierre Paradinas, Jean-Jacques Vandewalle, Patrick Biget, Patrick Georges, Jean-Louis Lanet et Benoît Brieussel. Sans oublier les autres équipes qui ont contribué à ce stage de manière indirecte.

Je tiens également à remercier Pierre Bruyère, Michel Escalant, Carole-Audrey Koch et Pascal Plouidy pour leur dévouement et leur patience.

Pour terminer, je voudrais remercier les personnes qui m'ont aidées dans la rédaction et la correction de ce document. Je remercie par conséquent M. Ramaekers et Pascal Goossens pour leurs nombreux conseils et le temps passé aux relectures, mais aussi Bianco Sylvie pour la correction des fautes d'orthographe et grammaticales.

Résumé

Les contraintes liées aux cartes à puce actuelles ont conduit les chercheurs de Gemplus à penser une nouvelle génération de carte à puce (carte générique) disposant d'un système d'exploitation nouveau et d'un nouveau système de sécurité.

Dans ce mémoire, nous essayons de présenter la carte générique et donnons un descriptif de cette carte et du nouveau mécanisme de sécurité mis en place. Nous n'aborderons pas l'aspect système d'exploitation mais nous nous focaliserons sur les aspects sécurité induits par les interactions entre la carte et son environnement. Nous expliquerons par conséquent le protocole réalisant cette sécurité et présenterons une application mettant en oeuvre le protocole.

Abstract

Many actual smart cards contraints (smart cards are static) lead researchers from Gemplus to think that a new generation of smart cards with a new operating system and thus a new view of the security must be issued.

In this these we try to explain what are the generic smart cards and what are their life cycle and what are the differences between a classic smart card and a generic smart card. We don't present the operating system of these cards but the new view of the security due to the fact that these cards are interacting with their environment. Thus we explain a protocol that realise the security and we present an application that implement this protocol.

Plan du mémoire

Table des figures

Table des tableaux

Introduction générale

Introduction générale

Sommaire

Ce mémoire, rédigé au cours de ces derniers mois, conclut un stage qui s'est déroulé au sein de la société GEMPLUS à Marseille. Il présente, outre des aspects déjà connus de la carte à puce, un nouveau type de carte appelée carte Combo. Il met également en évidence les exigences requises par la carte Combo en matière de sécurité.

Cette partie théorique sera complétée par une partie plus pratique qui présente l'implémentation, à l'aide de la librairie *Cryptoki* répondant au standard PKCS#11, du protocole d'accès des partenaires à la carte.

Avant d'entamer les différents aspects de cette nouvelle génération de carte, nous voudrions vous présenter brièvement la société GEMPLUS (cadre de travail) et le projet Combo. Un plan du mémoire terminera alors cette introduction générale.

Cadre de travail

GEMPLUS fut fondée en mai 1988. A cette époque, elle s'était établie provisoirement à Aix-en-Provence et employait 90 personnes. Aujourd'hui, elle emploie plus de 2500 personnes dans ses différents départements regroupés aux alentours de Marseille, avec le centre principal à Gemenos. Outre son implantation en Europe, nous la retrouvons aux Etats-Unis, en Asie et en Australie. Elle s'affirme donc aujourd'hui comme leader mondial de la carte à puce.

A ses débuts, cette société ne s'attelait qu'à la fabrication de cartes à puce. Mais bien vite les dirigeants se rendent compte des potentiels du marché et décident alors de développer des produits dans le domaine des cartes. Elle propose des équipements hardware, des logiciels produits ou applicatifs, des services liés à la carte, et en particulier la personnalisation afin que les cartes livrées aux clients constituent la solution idéale et parfaitement adaptée, intégrée avec le minimum d'effort dans l'application du client. C'est ainsi que nous retrouvons la société GEMPLUS dans des secteurs allant de la télécommunication aux transports en commun en passant par le secteur bancaire. Sans compter ses nombreuses alliances avec des collaborateurs d'autres domaines afin d'être présente sur de nouveaux marchés présents et futurs.

Le stage à partir duquel a été rédigé ce mémoire s'est déroulé dans le département de Recherche et Développement et plus particulièrement dans l'équipe de recherche dirigée par Pierre Paradinas. Cette équipe, qui emploie sept personnes, est chargée de développer le projet Combo que nous vous présentons ci-dessous.

Projet Combo

Le travail présenté dans ce mémoire s'inscrit dans le cadre d'un projet nommé Combo, développé par l'équipe de recherche de GEMPLUS en association avec des partenaires et d'autres départements de l'entreprise, dont l'objectif est d'implémenter un nouveau type de carte à puce.

De nombreuses contraintes liées aux cartes à puce ont conduit l'équipe de recherche à imaginer un nouveau type de carte accompagnée d'un nouveau système d'exploitation. Parmi ces contraintes, retenons le caractère statique des cartes : les fonctionnalités (routines et données) introduites dans une carte par un prestataire de services (institution autorisée à charger et décharger des fonctionnalités dans des cartes) et l'émetteur de cette carte ne pourront jamais être modifiées. On dit alors que la carte est dédiée à une application.

Le travail de cette équipe de recherche est alors de développer une carte à microprocesseur capable, au cours de son cycle de vie, de se voir charger et décharger de nouvelles fonctionnalités devenant disponibles pour les clients (institutions autorisées à exécuter, au travers d'applications, les fonctionnalités des cartes) sous la forme de services. Le client pourra alors développer des applications à l'aide de ces services. Cette carte sera la carte Combo.

Cette nouvelle vision de la carte à microprocesseur nécessite un renforcement du système de sécurité à deux niveaux : d'une part, une sécurité face à certains partenaires (notamment les prestataires de services et clients) entrant en communication avec la carte et d'autre part, une sécurité au niveau des routines et des données constituant un service.

Nos objectifs en ce qui concerne ce mémoire se limitent à un des deux niveaux de sécurité, en l'occurrence la sécurité face aux partenaires entrant en communication avec la carte. Nous vous proposons d'en découvrir les objectifs plus en détail au travers de la section ci-dessous (Objectifs du mémoire).

Objectifs du mémoire

Nous aimerions vous présenter, outre les objectifs assignés à ce stage, la manière qui nous a permis d'y arriver.

L'objectif du stage est d'implémenter un protocole permettant de limiter l'accès à la carte Combo à des partenaires autorisés. Un prototype mettant en présence une carte, un prestataire de services et un client, devait d'abord être réalisé afin d'en montrer la faisabilité à l'aide de la librairie Cryptoki répondant au standard PKCS#11. Une implémentation plus conviviale faisant intervenir plusieurs partenaires (prestataires de services et clients) devait être mise en place. Elle permettrait alors de montrer la faisabilité du protocole lorsque qu'une carte entre en contact avec ces partenaires.

La Figure 1 [PCO,97] présentée ci-dessous montre les différentes parties du projet Combo.

La partie en caractères gras sera implémentée dans ce mémoire. Les aspects propres au système d'exploitation ne seront pas abordés, ils ont fait l'objet d'une étude menée par Monsieur Jean-Jacques Vandewalle [JJV,97].

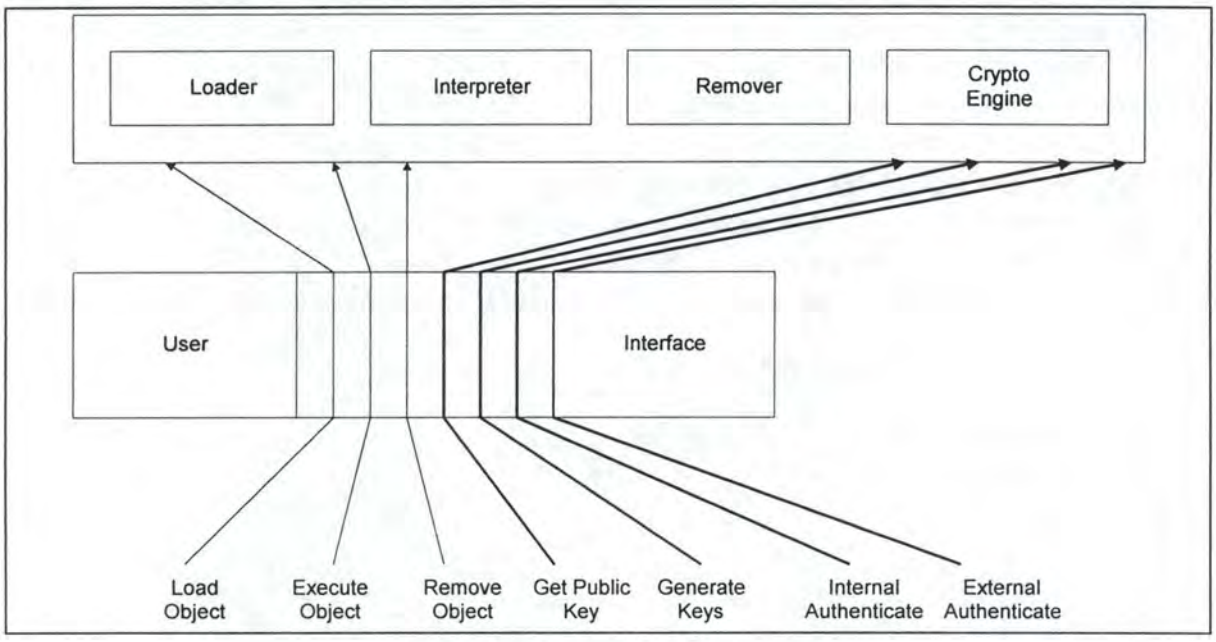


Figure 1 : Composantes du projet.

Bon nombre de matières ont d'abord dû être maîtrisées. Nous avons donc, avant d'entamer l'implémentation du protocole, préféré réaliser une étude de la carte à microprocesseur en générale et bien entendu de la carte Combo (carte générique). Ce qui nous a amené peu à peu à l'étude du protocole et de la cryptographie. Après une brève formation cryptographique et une étude de la librairie Cryptoki répondant au standard PKCS#11, nous avons pu débuter l'implémentation proprement dite.

Ce cheminement, nous vous proposons de le retrouver en détail dans la section ci-dessous consacrée à la description du plan du mémoire.

Plan du mémoire

Nous allons vous décrire dans cette section les différentes parties qui composent ce mémoire ainsi qu'un descriptif de leur contenu.

Ce travail se décompose en trois parties :

- La première partie, intitulée *Carte Combo*, regroupe trois chapitres. Dans le premier chapitre, nous détaillerons l'état de l'art de la carte à puce. Nous passerons alors en revue différents types de cartes à puce. Le deuxième chapitre traitera de la nouvelle génération de cartes à puce (carte Combo). Le troisième chapitre nous permettra d'introduire le protocole de sécurité implémenté au cours de ce mémoire.

- La deuxième partie, intitulée *Réalisation du Protocole*, se décompose également en trois chapitres. Le quatrième chapitre, après un rappel des objectifs, nous indiquera les outils utilisés lors de l'implémentation ainsi que la méthodologie suivie. Le cinquième chapitre sera consacré à la spécification du standard PKCS#11. Nous présenterons les fonctions de la librairie Cryptoki et nous reformulerons le protocole à partir de certaines de ses fonctions. Le sixième chapitre sera consacré à la spécification de l'application implémentant le protocole de sécurité.

- La dernière partie est intitulée *Tests et Exemples*. Elle nous permettra de montrer d'une part le fonctionnement du protocole au travers de quelques tests et d'autre part sa résistance à différentes attaques.
  Un exemple, illustrant un cas concret, sera repris au travers de quelques pages écran.

Partie 1 : La Carte Combo

Chapitre 1 : L'état de l'art des cartes

Sommaire

Ce chapitre est consacré à l'historique des cartes à puce. Il passe en revue différents types de cartes à puce en mettant l'accent plus particulièrement sur les cartes à microprocesseurs en indiquant leurs composants techniques et les deux types de cartes à microprocesseur.

1.1 La carte à mémoire

1.1.1 Historique

En 1974, Roland Moreno dépose un brevet relatif à une mémoire protégée personnelle permettant de réaliser des transactions de paiement. Ces idées ont, dans un premier temps, été concrétisées en France par des expériences menées sous le patronage du Groupement d'Intérêt Economique " carte à mémoire " regroupant les banques et France Telecom. Dès la fin des années 70, France Telecom soutient donc un pari ambitieux en choisissant la carte à mémoire comme moyen de paiement des communications (télécartes). Ces expérimentations aboutissent en 1983 au lancement de la téléphonie publique à carte.

1.1.2 Définition

La carte à mémoire est la carte la plus simple, elle ne stocke que de la valeur à consommer ou de l'information. Elle peut bénéficier soit d'un accès libre, où le porteur de la carte a accès au service quand il l'engage dans un lecteur, soit d'un accès contrôlé, où la mémoire est protégée par un code confidentiel. [GEM,97]

1.2 La carte à microprocesseur

1.2.1 Historique

En 1983, la carte à microprocesseur est créée. Ce n'est qu'en 1992 qu'elle s'implante sur le marché. Aujourd'hui, la carte à puce est présente dans les secteurs financiers (paiement électronique) mais également dans des applications militaires (contrôle des accès à des camps), dans l'automobile et dans bien d'autres domaines encore.

L'application de la télécarte dans le système de publiphonie de France Télécom et celle de la carte de crédit " Carte Bleue " sont certainement les plus grandes applications actuelles de la carte à puce.

Aujourd'hui, nous pouvons recenser de nombreux domaines où la carte fait l'objet de recherche. Parmi ceux-ci, nous pouvons citer la carte santé.

1.2.2 Définition

La carte à puce est une carte dite « astucieuse ». Elle ajoute à la zone mémoire des compétences informatiques pilotées par un microprocesseur qui se charge de garder l'accès et d'effectuer des opérations élaborées en travaillant conjointement avec les lecteurs auxquels il peut être connecté. [GEM,97]

1.2.3 Les composants technologiques

De la définition présentée ci-dessus, nous pouvons extraire deux composantes (Figure 1 - 1) de la carte :

- le microprocesseur,
- les mémoires.

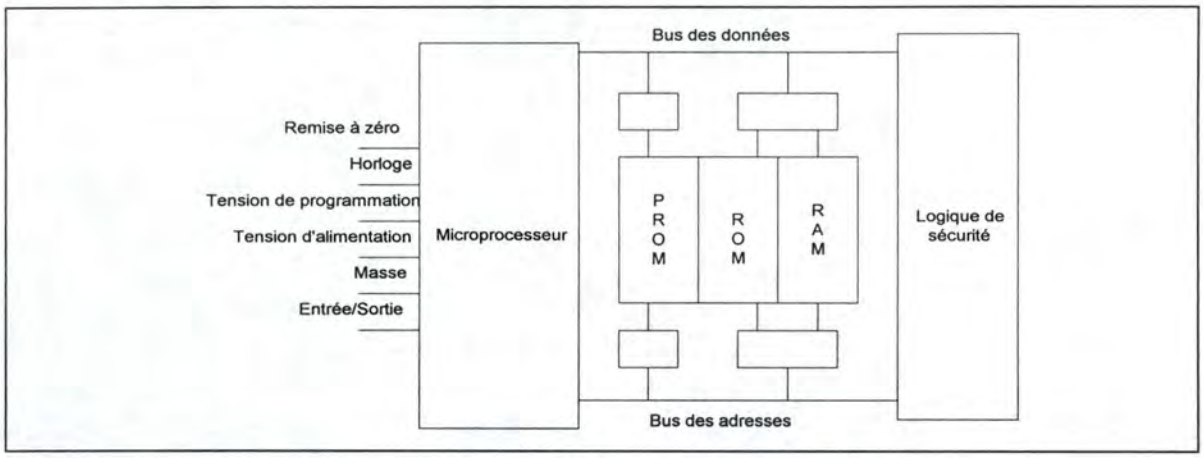


Figure 1 - 1 : Architecture d'une carte à microprocesseur.

- Le microprocesseur [RBR,88]
  Le microprocesseur (généralement un 8 bits), l'élément intelligent de la carte à puce, réalise deux types de fonctions de base : la manipulation et l'interprétation de données. Il réalise ces fonctions en exécutant des instructions stockées en mémoire. Un ensemble d'instructions constitue le programme.

  Nous pouvons distinguer deux types de programme : le système d'exploitation et les programmes d'application.

  - Le système d'exploitation « transforme un appareil physique en un outil logique en fournissant les spécificités suivantes :
  * Gestion de la mémoire,
  * Gestion de la sécurité,
  * Fonctions cryptographiques ». [GPK,96]

  Placé en ROM, il comprend différentes routines indépendantes de toute application particulière mais utilisables par des applications.

- **Les programmes d'application** comprennent l'ensemble des instructions qui définissent les fonctions que la carte doit remplir au cours d'une application spécifique.

- Les mémoires [RBR,88]
  Les mémoires utilisées dans les cartes à puce ont une capacité allant de 8 ko à 64 ko. Elles peuvent être divisées en trois types :

  - **Read Only Memory (ROM)**
    La ROM est une mémoire non volatile dont le contenu est stocké lors de la fabrication de la puce et ne peut plus être modifié.

  - **Random Access Memory (RAM)**
    La RAM est une mémoire volatile caractérisée par le fait que le système d'exploitation ou les programmes d'application peuvent y écrire, y effacer, y modifier et y lire des données. Elle sert généralement de zone de stockage temporaire.

  - **Programmable Read Only Memory (PROM)**
    La PROM est une mémoire non volatile et programmable. Il existe deux grandes catégories de mémoire PROM :
    * Electrically Programmable Read Only Memory (EPROM) :
      La EPROM est une mémoire qui n'est programmable qu'une seule fois et non effaçable.

    * Electrically Erasable Programmable Read Only Memory (EEPROM) :
      La EEPROM est une mémoire effaçable et programmable électriquement.

Outre ces deux composantes, la carte dispose également d'une logique de sécurité et bien entendu de contacts :

- La logique de sécurité
  La logique de sécurité a pour objectif de détecter tous types de violation de la carte aussi bien physique que logique. C'est ainsi qu'elle mesure, grâce à des capteurs, la résistance de l'enrobage de la puce. Si cet enrobage est violé par exemple en y perçant un trou, la logique de sécurité bloque alors la carte. Il en va de même si quelqu'un tente de retirer la couche protectrice de la puce ou si l'on augmente la vitesse de son horloge.

- Les contacts
  Les contacts sont au nombre de six. Nous pouvons distinguer l'horloge, la remise à zéro, la tension de programmation, la tension d'alimentation, la masse et les entrées/sorties.

1.2.4 Les types de cartes

Aujourd'hui, les cartes à puce disposent d'une architecture classique comme celle définie ci-dessus. Elles se différencient par leur taille de mémoire, leur vitesse de processeur et bien entendu par leur type de système d'exploitation et de programmes d'application.

Néanmoins, nous pouvons classer les cartes à puce en deux types :

- Carte à contacts
  Les cartes à contacts sont les cartes que nous rencontrons habituellement telles que Visa, Eurocard, MasterCard,...

- Carte sans contact
  Les technologies sans contact sont utilisées comme moyen de transférer des données entre une carte à puce et des appareils de lecture/écriture qui n'ont pas besoin d'utiliser des contacts extérieurs. Il n'y a aucun contact entre la carte et le lecteur. Le transfert de données de la carte vers un appareil de lecture/écriture se réalise par modulation d'amplitude où le signal envoyé varie en amplitude entre deux niveaux, un niveau représentant un 1 binaire et l'autre représentant un 0 binaire.

1.2.5 Les étapes de la vie d'une carte

Nous pouvons distinguer huit étapes caractérisant la vie d'une carte à puce :

- La fabrication de la puce :

  - **Sciage** :
    fractionnement de la tranche de silicium en puces,
  - **Collage** :
    positionnement, tenue mécanique et contact électrique de la puce sur le support film,
  - **Soudure des fils** :
    réalisation des connexions électriques,
  - **Encapsulation** :
    enrobage de l'ensemble de la puce dans une couche protectrice,
  - **Fraisage Décourt-Circuitage** :
    diminution de l'épaisseur du micromodule pour permettre l'opération d'encartage,
  - **Inspection visuelle finale** :
    détection des micromodules ayant un défaut visuel.

- La fabrication des cartes

  - **Moulage de la carte** :
    injection d'une matière plastique dans un moule afin d'obtenir un corps de carte avec une cavité de la taille de la puce,
  - **Impression Vernissage** :
    impression sur la carte d'un logo ou d'un graphisme suivie du recouvrement par un verni protecteur.

- L'encartage : collage de la puce dans la cavité de la carte.
- La personnalisation : impression graphique donnant une identité à la carte et chargement des informations propres au porteur.
- Les tests : tests électriques de la puce des cartes.
- La distribution et l'utilisation.
- La mort

Dans le chapitre suivant, ces huit étapes seront regroupées de manière à former les trois étapes du cycle de vie d'une carte :

- *la fabrication* reprenant la fabrication de la carte, la fabrication de la puce et l'encartage,
- *la personnalisation* reprenant l'impression graphique, le chargement des informations et les tests,
- *l'utilisation* reprenant la distribution, l'utilisation et la mort de la carte.

Nous compléterons chacune de ces trois phases au cours du chapitre suivant.

1.3 La carte multi-application

La carte multi-application, tout comme la carte à puce classique (dédiée à une application) dispose d'une architecture classique (Figure 1 - 1).

Disposant d'une mémoire plus importante que la carte classique, elle peut accueillir plusieurs routines et données fournissant ainsi différents services aux clients. Elle doit par conséquent, abriter un système d'exploitation particulier permettant de vérifier que les différents routines n'utilisent ou ne modifient pas les données appartenant à d'autres, que les routines ne se modifient pas mutuellement ou encore qu'ils ne tentent d'obtenir les secrets de la carte. Nous pensons notamment à la clé privée de la carte. Du point de vue de son système d'exploitation, la carte multi-application se rapproche de la carte générique que nous verrons dans le chapitre suivant.

Malheureusement, cette carte n'efface pas le caractère statique des cartes classiques. En effet, le système d'exploitation n'est pas prévu pour permettre des chargements et des déchargements de fonctionnalités. Son cycle de vie est par conséquent identique à celui d'une carte classique, comme nous le verrons dans la chapitre suivant. Cette carte est dédiée à plusieurs applications mais reste néanmoins statique. Les applications qu'elle contient ne pourront jamais être modifiées. De plus, de nouvelles applications ne pourront pas être chargées.

Chapitre 2 : La carte Combo

Sommaire

Avant d'entamer la présentation de cette carte, il faut signaler qu'à l'heure actuelle, elle n'est ni produite, ni en phase d'expérimentation. Nous pouvons dire qu'elle n'existe que sur papier. Les spécifications en sont établies mais nous n'en ferons pas référence dans ce mémoire. Nous devrons donc nous limiter à en donner une définition, les caractéristiques principales et son cycle de vie. [PCO,96] [JJV,97]

2.1 L'idée d'une carte générique

Bon nombre de contraintes liées aux cartes à puce ont conduit les membres de l'équipe recherche à spécifier un nouveau type de carte, dite carte Combo qui est une carte générique.

Tout d'abord, les cartes à puce actuelles sont statiques : les fonctionnalités sont chargées en ROM lors de l'étape de fabrication et ne peuvent jamais être mis à jour. De plus, le système d'exploitation et les fonctionnalités sont liées, ce qui oblige les prestataires de services à être totalement dépendant des fabricants lors de réalisation de leurs services.

L'idée est donc de fournir une carte où les services pourraient être chargés dynamiquement (selon les souhaits du porteur), conçues indépendamment du développement du système d'exploitation. Cela implique donc une structure de système d'exploitation totalement nouvelle comportant d'une part un environnement de développement de services pour le prestataire de services et d'autre part certains concepts permettant de faciliter l'intégration de la carte dans des systèmes d'information orientés objets.

Partant d'une carte vierge de toute application, le porteur pourra demander aux prestataires de services le chargement des applications qu'il désire utiliser. Pour le prestataire de services, il s'agit alors de charger les services à partir desquels le client a réalisé les applications souhaitées par le porteur. Au cours de sa vie, la carte subira donc différents chargements et déchargements de services.

L'intérêt d'une telle carte est qu'elle n'est pas dédiée dès son origine à une seule application comme c'est le cas actuellement pour les cartes à puce classiques. Cela implique par conséquent de revoir le cycle de vie des cartes à puce actuelles afin qu'il évolue vers un cycle de vie adapté à la carte générique.

Il fut donc décidé que la carte Combo disposerait d'une architecture classique accueillant un système d'exploitation particulier (Figure 2 - 1).

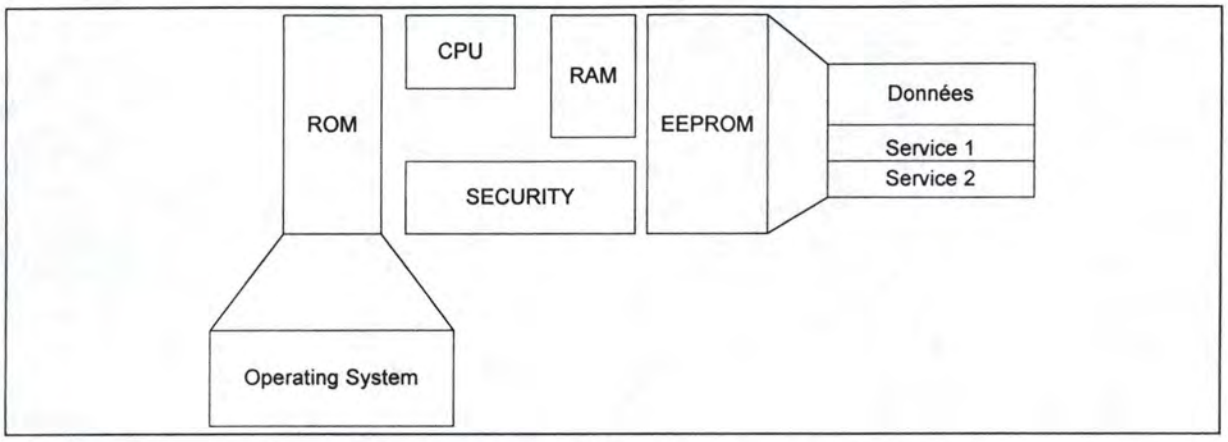


Figure 2 - 1 : Architecture d'une carte à puce générique.

2.2 Définition d'une carte générique

Nous appelons carte générique « une carte à microprocesseur capable au cours de son cycle de vie de charger et décharger de nouvelles fonctionnalités devenant disponibles pour les clients sous la forme de services ». [JJV,97]
Ces fonctionnalités sont regroupées sous la forme de services composés de données et de routines effectuant des traitements sur ces données.

2.3 Les partenaires de la carte générique

Au cours de sa vie, une carte va entrer en relation avec différents partenaires. Nous pouvons distinguer cinq catégories de partenaires intervenant au cours du cycle de vie (Figure 2 - 4) de la carte à puce dont les relations sont représentées à la Figure 2 - 2.

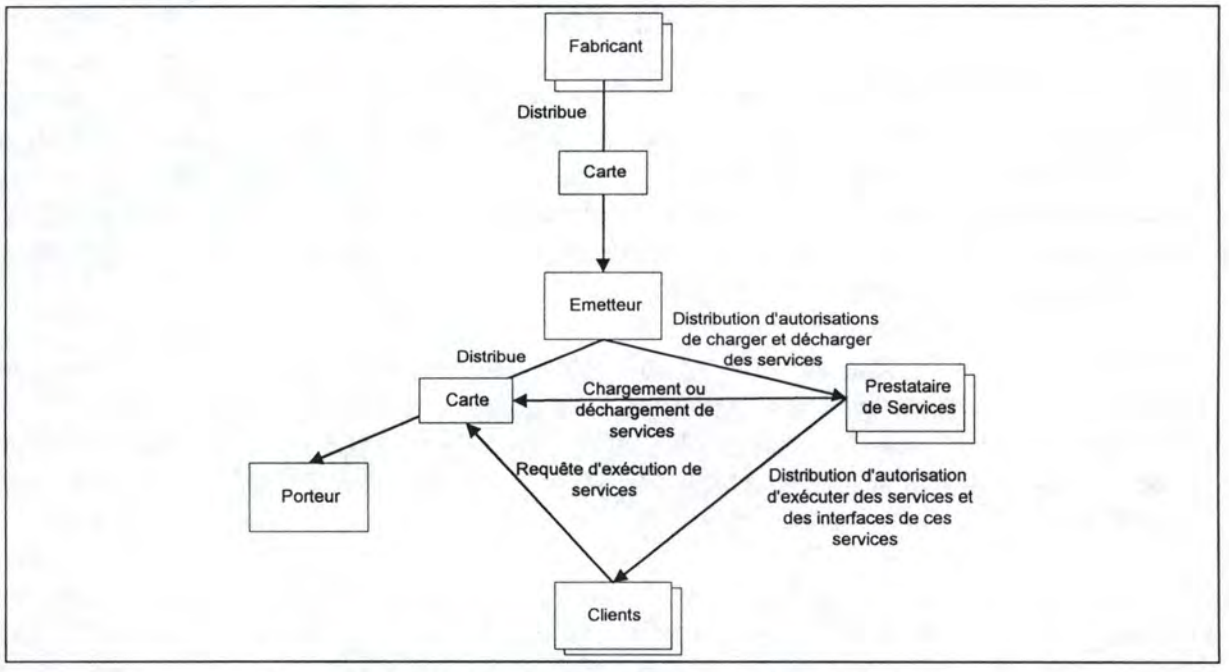


Figure 2 - 2 : Relations entre les partenaires de la carte.

Chaque partenaire joue un rôle précis au cours de ses relations avec la carte. Détaillons par conséquent chacun de ces rôles :

Le fabricant
Il intègre le microprocesseur sur le support plastique et charge le système d'exploitation dans la carte. Le système d'exploitation offre des fonctions de chargement, de déchargement, d'exécution de programmes et est également chargé de la sécurité.

L'émetteur
Il est l'autorité responsable de la délivrance de la carte au porteur. Il effectue la personnalisation de la carte et s'occupe de la distribution des autorisations (accréditations ou certificats) de charger et décharger des services aux prestataires de services. La personnalisation lie la carte à son porteur.

Le prestataire de services
Il charge des services dans la carte à la demande du porteur et après avoir prouvé qu'il est autorisé par l'émetteur. Il distribue également, aux clients des services, des autorisations d'utiliser ces services. Il fournit des interfaces d'accès permettant aux clients de réaliser des applications utilisant les services.

Les clients
Ils sont les utilisateurs finaux des services chargés dans la carte. Ils développent des applications utilisant les services d'une carte à l'aide des interfaces reçues des prestataires. Les services sont exécutés dans la carte, grâce aux applications développées par les clients, dès qu'elle a vérifié que le client est autorisé à les exécuter.

Le porteur
C'est l'utilisateur de la carte. A ce titre, il est le seul à pouvoir demander à des prestataires de services que des services soient chargés dans la carte ou déchargés de la carte.

2.4 Les cycles de vie

2.4.1 Présentation

Comme nous l'avons signalé précédemment, la carte classique et la carte générique ont des cycles de vie différents. Néanmoins, qu'il s'agisse d'une carte classique ou d'une carte générique, leur cycle de vie, comme nous l'avons expliqué dans le chapitre précédent, se compose de trois phases : *la fabrication*, *la personnalisation* et *l'utilisation*. Complétons à présent ce qui a été décrit dans le chapitre précédent en distinguant la carte à puce classique de la carte à puce générique.

Pour la carte classique (Figure 2 - 3) :

- Au cours de *l'étape de fabrication* (non représentée sur la figure ci-dessous), le système d'exploitation ainsi que les fonctionnalités (devenant disponible pour le client sous forme de services) sont placés en ROM.

- Au cours de *l'étape de personnalisation*, l'émetteur et le prestataire de services peuvent ajouter certaines fonctionnalités sous la forme de programmes filtres.

- Au cours de *l'étape d'utilisation*, la carte ne fait que répondre à des demandes d'exécution de services qui ont été chargés à l'étape de fabrication. Aucun nouveau service ne pourra être chargé dans la carte.

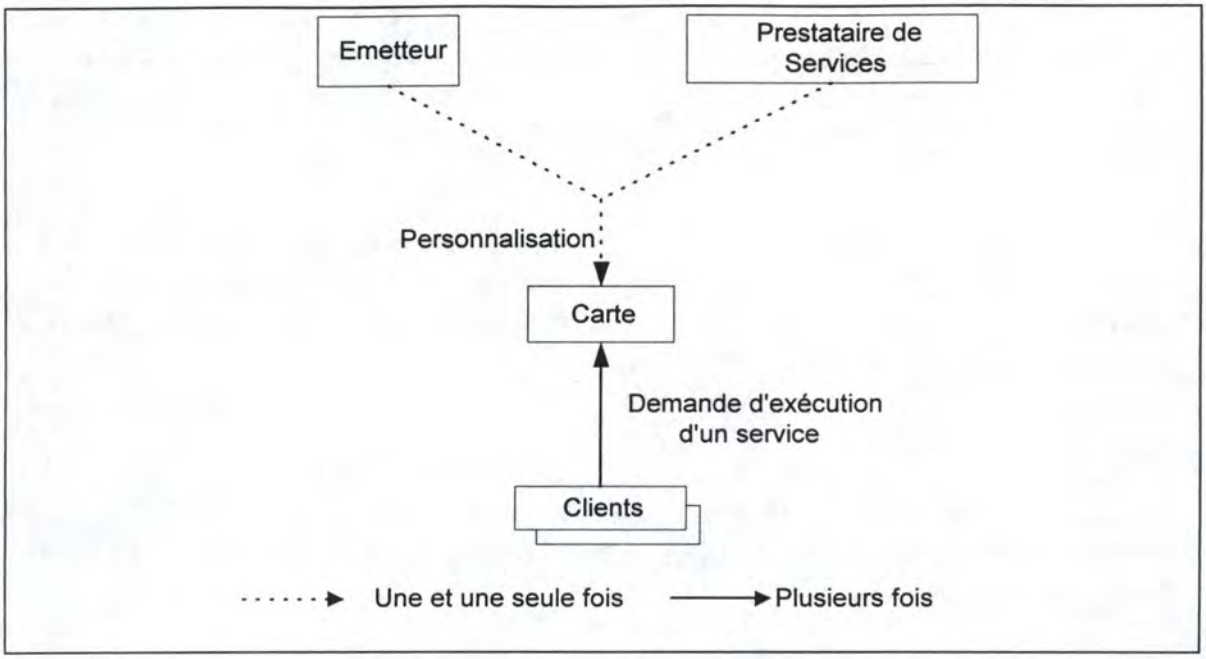


Figure 2 - 3 : Cycle de vie d'une carte classique.

Pour la carte générique (Figure 2 - 4) :

- Au cours de *l'étape de fabrication* (non représentée sur la figure ci-dessous), le système d'exploitation et les fonctionnalités sont chargés dans la carte et constituent le noyau minimum permettant plus tard d'accueillir de nouvelles fonctionnalités.

- Au cours de *l'étape de personnalisation*, l'émetteur peut ajouter de nouvelles fonctionnalités, soit en étoffant le noyau de base, soit en utilisant une fonction de chargement de services.

- Au cours de *l'étape d'utilisation*, les prestataires de services vont charger de nouveaux services ou en décharger d'anciens. La carte ne fait que répondre à des demandes d'exécution de services qui ont été chargés.

L'émetteur distribue également des autorisations de charger et décharger des services dans la carte. Chaque nouveau prestataire de services se faisant connaître auprès de l'émetteur recevra cette autorisation.

Les prestataires de services auront distribué aux clients l'interface des services ainsi que les autorisations d'utiliser les services placés dans la carte.

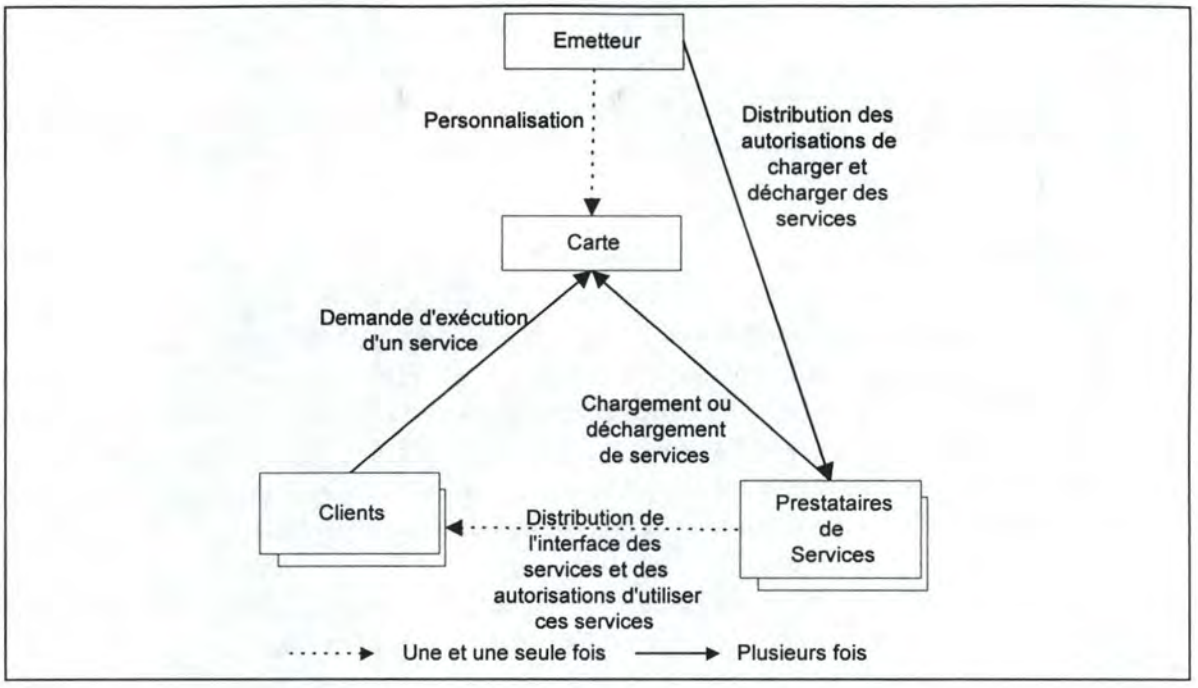


Figure 2 - 4 : Cycle de vie d'une carte générique.

Le niveau de fonctionnalités de la carte à puce générique va donc évoluer au cours de la phase d'utilisation comme nous pouvons le constater à la Figure 2 - 5.

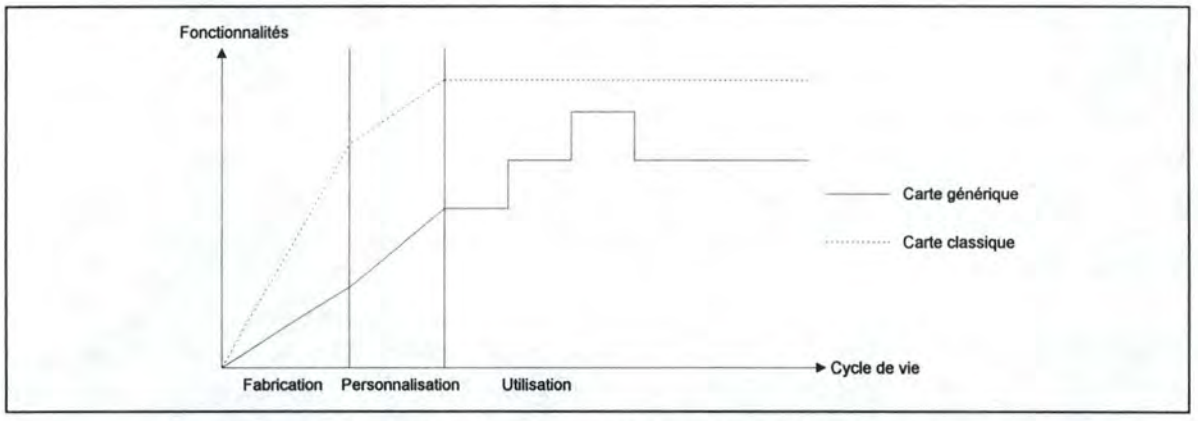


Figure 2 - 5 : Evolution du niveau de fonctionnalités des cartes.

2.4.2 Les impacts induits par le cycle de vie de la carte générique

L'impact induit par ce nouveau type de carte est triple et se situe tant au niveau de la sécurité qu'au niveau du développement d'applications.

Le premier impact concerne la sécurité de la carte face à ses partenaires, en l'occurrence les prestataires de services (institution autorisée à charger et décharger des fonctionnalités dans des cartes) et les clients (institutions autorisées à exécuter, au travers d'applications, les fonctionnalités des cartes). La carte ne connaissant pas à l'avance ses partenaires, va devoir s'assurer de leur droit à effectuer une ou plusieurs tâches particulières lorsqu'ils se présenteront à elle. L'idée est alors de créer un mécanisme de distribution d'autorisations.

La Figure 2 - 4 nous montre alors que par ces mécanismes, les prestataires de services sont sous l'autorité de l'émetteur qui leur distribue les autorisations.
Les clients sont quant à eux sous l'autorité des prestataires de services qui leur distribuent les autorisations. Il faut bien entendu que la carte dispose d'un mécanisme de contrôle de ces autorisations.

Le deuxième impact concerne les routines et données, constituant des services stockés dans la carte. A chaque routine correspondent des données sur lesquelles il effectue un traitement particulier afin de réaliser un service. La préoccupation principale est d'assurer une sécurité entre tous ces routines et données. En effet, il faut veiller à ce qu'aucun routine n'utilise les données d'un autre, qu'aucun routine n'écrase les données d'un autre et qu'aucun routine n'en écrase un autre. De plus, il faut veiller à ce qu'aucun routine ne prenne la main sur le système d'exploitation. La carte doit par conséquent disposer d'un environnement d'exécution sécurisé afin d'éviter tous ces désagréments.

Le dernier impact concerne les clients qui doivent pouvoir développer facilement des applications utilisant les services. Il faut donc fournir des méthodes de développement d'applications carte.

Nous ne nous intéresserons pas dans ce mémoire aux deux derniers impacts mais développerons en détails le premier impact lors du chapitre suivant, et en présenterons une solution implémentée.

2.4.3 Particularités dues au cycle de vie de la carte générique

Deux particularités se dégagent de ce cycle de vie. D'une part, l'ensemble des prestataires de services et clients n'est pas connu. Un nouveau prestataire de services peut se faire connaître auprès d'un émetteur afin d'obtenir une autorisation de charger des services dans les cartes de cet émetteur. Un nouveau client peut être autorisé par un prestataire de services à utiliser un service chargé dans des cartes. D'autre part, l'ensemble des services de la carte est inconnu.

Ces particularités renforcent le besoin pour la carte de pouvoir disposer d'un mécanisme de contrôle des partenaires qui se présentent à elle. Nous vous proposons de retrouver ce mécanisme au cours du chapitre suivant.

Chapitre 3 : Les accès à la carte Combo

Sommaire

Au cours de la présentation de la carte Combo (chapitre 2), nous avons mis en évidence les impacts induits par le cycle de vie d'une telle carte au niveau de la sécurité. Comme nous l'avons dit, le premier impact a retenu toute notre attention et sera développé dans ce mémoire. Il concerne un mécanisme de protection de la carte face aux accès des différents partenaires. Ce mécanisme de protection est mis en oeuvre au travers d'un protocole cryptographique dit d'accès des partenaires à la carte.

Nous avons jugé opportun de présenter brièvement différents mécanismes cryptographiques au cours d'une première section (3.1 Les mécanismes cryptographiques).

Dans le chapitre 2, nous avons identifié les partenaires en relation avec la carte. Il nous reste maintenant à identifier, parmi ces partenaires ceux qui vont directement intervenir dans le protocole d'accès des partenaires à la carte. C'est ce que nous nous proposons de vous présenter au cours de la deuxième section (3.2 Les partenaires du protocole).

3.1 Les mécanismes cryptographiques

3.1.1 Définition

La cryptographie est l'un des moyens les plus utilisés aujourd'hui pour assurer l'échange ou la conservation sécurisée d'informations.

La cryptographie assure la confidentialité des données mais elle permet également [BSH,96]

- l'authentification : il doit être possible pour le récepteur d'un message d'avoir la certitude de sa provenance. Un intrus ne doit pas pouvoir se faire passer pour celui qu'il n'est pas.

- l'intégrité : il doit être possible pour le récepteur d'un message de vérifier qu'il n'a pas été modifié durant la transmission. Un intrus ne doit pas pouvoir remplacer un message donné par un autre.

- la non-répudiation : l'expéditeur ne doit pas pouvoir prétendre plus tard qu'il n'a pas envoyé un message.

3.1.2 Systèmes cryptographiques

Un algorithme cryptographique est une fonction mathématique utilisée pour le chiffrement et le déchiffrement. Historiquement, un tel algorithme était tenu secret par un groupe d'individu l'utilisant dans leur relation. L'inconvénient d'un tel système est que lorsqu'un individu quitte le groupe, les autres, pour des raisons de sécurité, sont obligés de rechercher un autre algorithme. Pour pallier ce problème, la cryptographie moderne a introduit le concept de clé. Une clé est alors définie comme un nombre de grande taille (512, 768 ou 1024 bits) et sera un des paramètres utilisés par les fonctions de chiffrement et déchiffrement.

Dans la littérature [BSH,96] et [JRA,97], nous pouvons distinguer les systèmes symétriques des systèmes asymétriques.

Système cryptographique symétrique

Un système cryptographique symétrique ou système cryptographique symétrique à clé secrète (Figure 3 - 1) est un système dans lequel la clé de chiffrement k utilisée par la fonction de chiffrement E (algorithme de chiffrement) est identique à la clé de déchiffrement utilisée par la fonction de déchiffrement D (algorithme de déchiffrement). Cette clé doit donc être gardée secrète par les individus l'utilisant.

Par conséquent,

$E_k(m) = m'$ est le message chiffré par la fonction E à l'aide de la clé k,

$D_k(m') = m$ est le message déchiffré (message en clair) par la fonction D à l'aide de la clé k.

Nous pouvons donc en déduire l'égalité que voici : $D_k(E_k(m)) = m$.

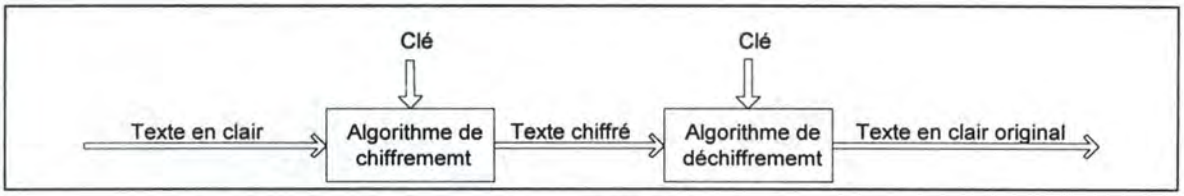


Figure 3 - 1 : Système cryptographique symétrique.

Le DES (Data Encryption Standard) est un algorithme de chiffrement - déchiffrement qui met en oeuvre ce système cryptographique (voir Annexe A0).

Système cryptographique asymétrique

Un système cryptographique asymétrique ou système cryptographique à clé publique (Figure 3 - 2) est un système dans lequel la clé de chiffrement *ke* (clé publique) utilisée par la fonction de chiffrement *E* (algorithme de chiffrement) est différente de la clé de déchiffrement *kd* (clé secrète), utilisée par la fonction de déchiffrement *D* (algorithme de déchiffrement). Les deux clés sont liées l'une à l'autre de façon telle que tout message chiffré avec *ke* ne puisse être déchiffré qu'avec *kd*. De plus, la clé de déchiffrement *kd* ne peut pas être calculée, dans un temps raisonnable, à partir de la clé de chiffrement *ke*.

L'algorithme de chiffrement est connu de tous tandis que l'algorithme de déchiffrement n'est connu que de celui qui possède de la clé secrète *kd*.

Par conséquent,

$E_{ke}(m) = m'$ est le message chiffré par la fonction E à l'aide de la clé *ke*,

$D_{kd}(m') = m$ est le message déchiffré par la fonction D à l'aide de la clé *kd*.

Nous pouvons donc en déduire l'égalité que voici : $D_{kd}(E_{ke}(m)) = m$.

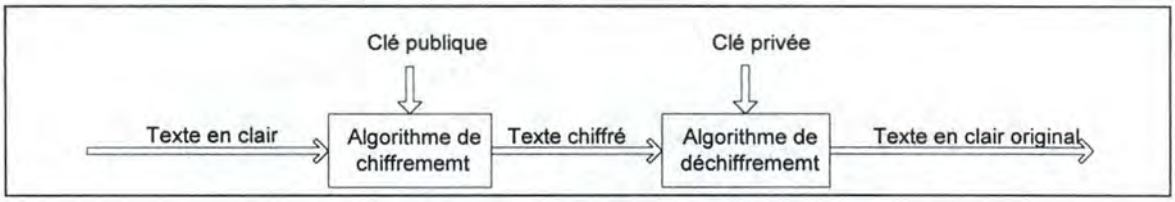


Figure 3 - 2 : Système cryptographique asymétrique.

Un tel système permet également de réaliser des signatures et des vérifications.

Un message *m* sera signé avec une fonction *S* et sera vérifié avec une fonction *V* (Figure 3 - 3). La signature s'effectue à l'aide de la clé secrète *kd* tandis que la vérification s'effectue à l'aide de la clé publique *ke*.

Par conséquent,

$S_{kd}(m) = m'$ est le message signé par la fonction S à l'aide de la clé *kd*,

$V_{ke}(m') = m$ est le message vérifié par la fonction V à l'aide de la clé *ke*.

Nous pouvons donc en déduire l'égalité que voici : $V_{ke}(S_{kd}(m)) = m$.

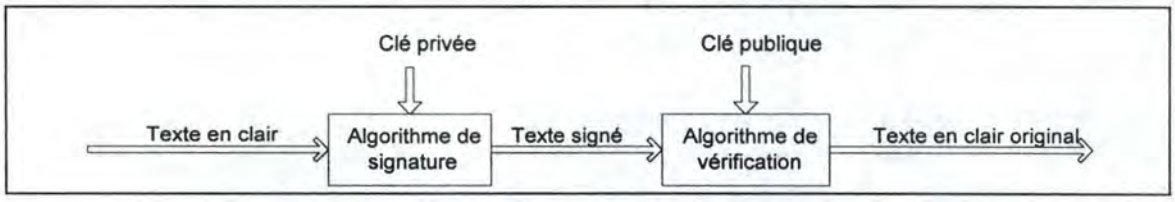


Figure 3 - 3 : Signature et vérification.

Le RSA (voir Annexe A1) est un algorithme de chiffrement-déchiffrement et signature-vérification qui met en oeuvre ce système cryptographique.

Le DSA (voir Annexe A2) est un algorithme qui met en oeuvre ce système cryptographique. Mais, il n'est utilisé que pour de la signature et vérification et non pour du chiffrement et du déchiffrement.

Nous pouvons également citer l'algorithme de Diffie-Hellman (voir Annexe A3) qui met en oeuvre ce système cryptographique.
Dans un système cryptographique asymétrique, la confidentialité est assurée grâce au mécanisme de chiffrement - déchiffrement. Quant à l'intégrité, la non-répudiation et l'authentification, elles peuvent être assurées par le mécanisme de signature - vérification.

Avant d'entamer la présentation du protocole mettant en oeuvre les concepts que nous venons de définir, examinons les partenaires de la carte intervenant dans le protocole.

3.2 Les partenaires du protocole

Des cinq partenaires de la carte que nous avons vus dans le chapitre précédent, tous ne vont pas intervenir dans le protocole d'accès. Passons en revue ces cinq partenaires et déterminons les intervenants au protocole.

Le porteur devra s'identifier auprès de la carte grâce au *Personal Identification Number* (PIN). Nous considérons que cette identification ne fait pas partie du protocole car, ce que nous voulons protéger, ce n'est pas directement le porteur de la carte vis-à-vis des voleurs mais bien les secrets qu'elle renferme. D'autre part, nous ne voulons pas non plus que certains partenaires se fassent piéger par des cartes non autorisées à être utilisées. Par conséquent, ce protocole tend à protéger aussi bien la carte que les partenaires entrant en communication avec elles. Le porteur est donc exclu de ce protocole.

Le fabricant est bien entendu à exclure également, car après fabrication, il n'intervient plus dans la vie de la carte.

L'émetteur est considéré comme une autorité de confiance qui, comme nous le verrons joue un rôle clé dans le protocole. Etant une autorité de confiance, aucune mesure visant une authentification ne sera réalisée.

Le prestataire de services va jouer un rôle dans le protocole : la carte lui impose de s'authentifier afin d'exclure tout faussaire qui tenterait d'y introduire des services. La carte s'authentifie également auprès de ce prestataire pour éviter qu'un faussaire n'obtienne des services chargés sur une fausse carte. Le protocole réalisera donc une authentification mutuelle (carte - prestataire de services).

Le client va également jouer un rôle important : la carte lui impose de s'authentifier afin d'exclure tout faussaire qui tenterait de s'y introduire afin d'exécuter des services. De plus, la carte s'authentifie également auprès de ce client pour éviter qu'un faussaire ne puisse utiliser une fausse carte. Le protocole réalisera donc une authentification mutuelle (carte - client).

Maintenant que nous connaissons les partenaires qui interagissent avec la carte, décrivons le protocole de sécurité des partenaires à la carte.

3.3 Le protocole de sécurité

Le protocole est composé de deux parties : la première partie comprend les accréditations des partenaires du protocole tandis que la seconde partie explique comment se déroulent les authentifications. Ce protocole se base sur un système cryptographique asymétrique.

3.3.1 Définition

Un protocole est « une série d'étapes, impliquant une ou plusieurs parties dans le but de réaliser une tâche ». [BSH,96]

Un protocole a les caractéristiques suivantes :

- Toute partie impliquée dans le protocole doit connaître le protocole et toutes les étapes à suivre à l'avance.
- Toute partie impliquée dans le protocole doit être d'accord de le suivre.
- Le protocole doit être non ambigu; chaque étape doit être bien définie et la probabilité de mauvaise compréhension doit être nulle.
- Le protocole doit être complet; à chaque situation possible doit correspondre une action spécifiée.

3.3.2 Accréditation des partenaires du protocole

Chaque partenaire du protocole (carte, prestataire de services, client et émetteur) possède une paire de clés : une clé publique et une clé secrète qui seront utilisées lors des chiffrements et déchiffrements.

Lors de la demande d'accréditation (l'octroi d'un certificat et d'une clé publique par une autorité), le demandeur reçoit une autre clé publique et un certificat (document électronique utilisé par un acteur pour prouver qu'il est autorisé à réaliser une action particulière). Afin d'expliquer la provenance de ces deux données, distinguons le cas du prestataire de services et de la carte de celui du client.

3.3.2.1 Prestataire de Services

L'accréditation appelée également certification d'un prestataire de services se déroule selon le schéma présenté à la Figure 3 - 4.

L'émetteur de la carte donne sa clé publique au prestataire de services ainsi qu'un certificat, ce qui constitue l'accréditation. Celle-ci est réalisée à chaque fois qu'un nouveau prestataire de services le désire mais ne peut se faire qu'une et une seule fois auprès d'un même émetteur. Le prestataire de services est alors enregistré comme étant autorisé à charger des services dans la carte.

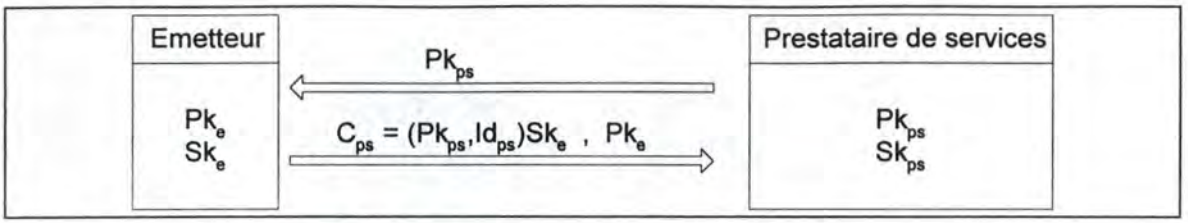


Figure 3 - 4 : Accréditation d'un prestataire de services.

Tout prestataire de services désirant charger des services dans des cartes doit se faire connaître auprès des émetteurs de ces cartes. Prenons l'exemple d'un prestataire et d'un émetteur de cartes. Le prestataire afin d'obtenir une « autorisation » de charger des services dans la carte, expédie sa clé publique à l'émetteur de la carte qui détermine si ce prestataire a déjà obtenu une « autorisation ». Si ce n'est pas le cas, l'émetteur crée alors l'autorisation du prestataire qui est un certificat composé de deux parties : la clé publique du prestataire (Pk_{ps}) et un identifiant (Id_{ps}). Ce certificat est signé à l'aide de la clé secrète de l'émetteur (Sk_e) et envoyé avec la clé publique de l'émetteur vers le prestataire de services.

Ce certificat n'est, en tant que tel, d'aucune utilité pour le prestataire de services. Mais, dans le cas de l'authentification (sous-section 3.3.3), il apportera la preuve à la carte (sous certaines conditions) qu'il est autorisé à effectuer des chargements ou des déchargements de services.

Le certificat est signé avec une clé secrète (en l'occurrence ici, la clé de l'émetteur) pour des raisons de sécurité. En effet, lorsque, lors d'une authentification, une carte recevra ce certificat d'un prestataire de services, de part le fait qu'il est signé, elle aura la certitude qu'il est vrai.

Il est bien entendu que par ce mécanisme tout le monde peut déchiffrer des certificats mais personne, mis à part l'émetteur ne pourra en créer.

Cette accréditation est réalisée au travers d'un échange sécurisé au cours duquel aucun autre partenaire n'a accès aux informations échangées.

3.3.2.2 Carte

Le mécanisme d'accréditation d'une carte, présenté ci-dessous, se base sur la Figure 3 - 5.

La carte reçoit tout comme le prestataire de services un certificat (C_c) créé par son émetteur, ainsi que la clé publique de l'émetteur. Le certificat contient le même type de données que celui du prestataire de services.

Par contre, la carte ne fait pas de démarche auprès de son émetteur afin d'obtenir une « identité », celle-ci lui est donnée lors de la phase de personnalisation. Au cours de cette phase, les clés de la carte sont inscrites en ROM. La clé secrète est protégée par un PIN (Personal Identification Code) différent de celui du porteur de la carte et connu par l'émetteur qui réalise la personnalisation.

Une carte ne reçoit qu'un seul certificat.

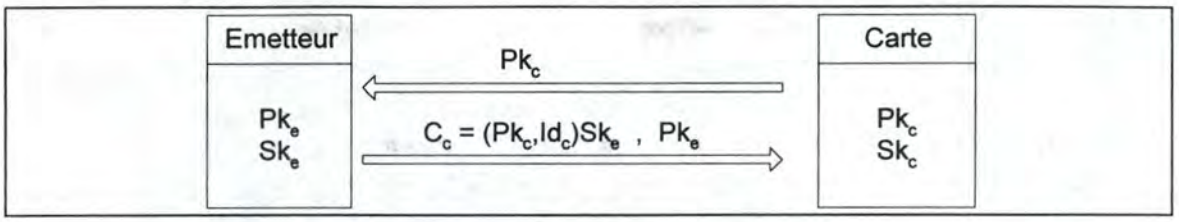


Figure 3 - 5 : Accréditation d'une carte

3.3.2.3 Client

L'accréditation d'un client (Figure 3 - 6) est différente des deux accréditations exposées ci-dessus. Tout d'abord, le client ne reçoit pas son certificat de l'émetteur de la carte mais d'un prestataire de services.

Un client peut faire la demande auprès de plusieurs prestataires de services dans le but d'obtenir des certificats. Mais, il ne pourra faire la demande qu'une seule fois auprès de chaque prestataire de services.

La raison de ces multiples demandes provient du fait qu'un client souhaite pouvoir exécuter des applications placées sur un certain nombre de cartes différentes c'est-à-dire des cartes émises par des émetteurs différents.

Le certificat d'un client est différent de celui des deux précédents partenaires du protocole. Il se compose en effet, outre de sa clé publique (Pk_{cl}), de deux nouvelles composantes : l'Object Reference et le Mask (suite de bits permettant à la carte de déterminer à quels services un client a accès). Tous deux vont limiter l'accès du client à certains services offerts par la carte. Le certificat est signé avec la clé secrète du prestataire de services (Sk_{ps}).

Le prestataire de services fournit également son identifiant au client et la clé publique de l'émetteur qui l'a certifié.

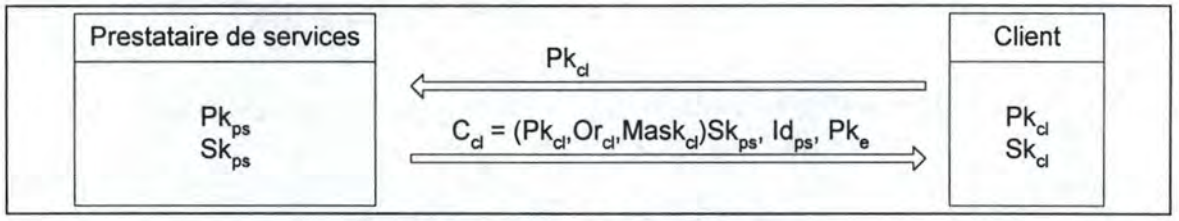


Figure 3 - 6 : Accréditation d'un client.

Il est important de signaler qu'un prestataire de services peut accréditer un client même si ce prestataire ne s'est pas authentifié auprès d'une ou plusieurs cartes et donc n'a encore chargé aucun service.

3.3.3 Authentification des partenaires du protocole

Une fois les accréditations effectuées, les partenaires du protocole sont en possession de la preuve de leur identité.

Ils vont donc utiliser ces certificats afin d'apporter la preuve à leurs interlocuteurs, en l'occurrence un autre partenaire du protocole, qu'ils sont bien ceux qu'ils prétendent.

Nous allons maintenant examiner comment se déroule le protocole d'authentification. Nous distinguerons l'authentification d'une carte et d'un prestataire de services, de l'authentification d'une carte et d'un client. Tous deux reposent sur une double vérification afin d'éviter qu'un partenaire du protocole ne se fasse passer pour un autre.

3.3.3.1 Carte - Prestataire de Services

Pour vous exposer cette première authentification, nous nous basons sur la Figure 3 - 7 [PCO,96].

Un prestataire de services, sur demande du porteur d'une carte, voudrait y charger un service. Ce prestataire prétend à la carte qu'il est accrédité à charger un service. Il va donc devoir prouver ses dires à la carte.

Dans le cas où le prestataire est réellement celui qu'il prétend, il souhaiterait ne pas se faire piéger par un éventuel faussaire qui tenterait d'obtenir un service. Pour se faire, il va demander à la carte si elle est bien accréditée à ce voir charger ce service. Une authentification mutuelle est donc souhaitée.

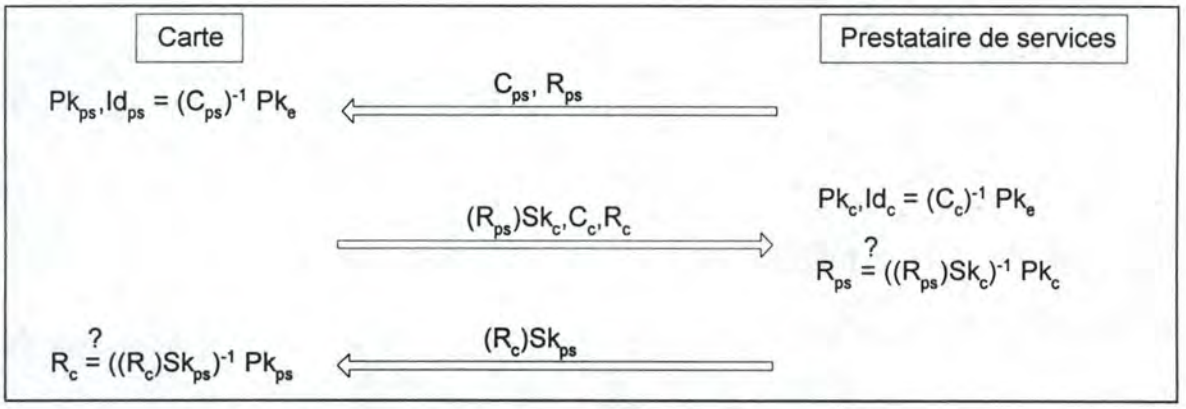


Figure 3 - 7 : Authentification d'un prestataire de services.

Nous allons découper le protocole en trois phases, chacune d'elles correspondant à une flèche sur la Figure 3 - 7.

Prestataire de services → Carte : C_{ps}, R_{ps}

Le prestataire de services va envoyer son certificat (C_{ps}) à la carte ainsi qu'un nombre aléatoire (R_{ps}).

Le certificat va permettre à la carte d'obtenir des informations sur le prestataire de services. En effet, la carte dispose de la clé publique de son émetteur et peut donc déchiffrer le certificat qu'elle vient de recevoir.

Dans le cas où le prestataire de services n'est pas accrédité, le certificat qu'il transmet à la carte n'est pas signé par le même émetteur que la carte. Le protocole est alors arrêté.

Maintenant, il se peut qu'un prestataire de services non accrédité dispose d'un certificat délivré par le même émetteur que la carte. Ce certificat, il peut l'avoir obtenu en espionnant un échange précédent entre un bon prestataire de services et la carte. Nous verrons la solution que propose le protocole au cours de la troisième étape.

Donc, la carte a reçu le certificat du prestataire de services et a vérifié que la signature était bien celle de son émetteur à l'aide de la clé publique de l'émetteur (Pk_e).

Le prestataire de services a également envoyé un nombre aléatoire (R_{ps}). Il va servir à prouver au prestataire de services que la carte qui va lui envoyer son certificat est réellement celle qui a reçu le nombre aléatoire et pas un tiers partenaire qui effectuerait un rejeu (réutilisation d'informations échangées au cours d'une communication antérieure entre deux interlocuteurs).

Au terme de cette étape, la carte ne sait encore rien quant au prestataire de services avec qui elle communique.

Carte → Prestataire de services : $(R_{ps})Sk_c$, C_c, R_c

La carte va maintenant prouver au prestataire de services qu'elle est accréditée par le même émetteur que lui. Pour ce faire, elle lui envoie son certificat (C_c). Elle envoie également un nombre aléatoire (R_c) qui sera utilisé lors de la troisième étape.

Le prestataire de services souhaite être sûr que la carte qui lui envoie ce certificat est réellement accréditée et n'est pas une carte qui a obtenu le certificat en espionnant d'autres échanges. Par conséquent, la carte va devoir signer avec sa clé secrète (Sk_c) le nombre aléatoire (R_{ps}) qu'elle a reçu, et l'envoyer au prestataire de services.

Le prestataire de services va alors vérifier à l'aide de la clé publique de l'émetteur (Pk_e) que la signature du certificat est bien celle de l'émetteur et ainsi obtenir de l'information sur la carte (clé publique et identifiant). Si cette signature n'est pas correcte le protocole est arrêté.

Il va maintenant déterminer que la carte dont il vient de recevoir le certificat est réellement accréditée. Pour ce faire, il vérifie à l'aide de la clé publique de la carte (Pk_c) la signature du nombre aléatoire qu'il a reçu et vérifie surtout que ce nombre correspond à celui qu'il a envoyé à l'étape précédente.

Si la signature est correcte et que les deux nombres aléatoires sont identiques, le prestataire de services a authentifié la carte.

Prestataire de services → Carte : $(R_c)Sk_{ps}$

Au cours de la première étape la carte a vérifié le certificat du prestataire de services et a pu ainsi obtenir sa clé publique (Pk_{ps}). Mais elle veut, avant de lui donner la permission d'effectuer des opérations, avoir la certitude qu'il est bien accrédité par l'émetteur. Pour se faire, le prestataire de services va devoir signer le nombre aléatoire (R_c) qu'il a reçu de la carte à l'aide de sa clé secrète (Sk_{ps}) et lui envoyer.

La carte recevant ce nombre signé, va vérifier la signature à l'aide de la clé publique du prestataire de services (Pk_{ps}). Si la signature est incorrecte le protocole est arrêté. Si la signature est correcte, le nombre aléatoire qu'elle récupère doit être identique à celui qu'elle a envoyé à l'étape précédente. Et dans ce cas, la carte a authentifié le prestataire de services.

Au terme de cette authentification mutuelle, la carte conserve l'identifiant du prestataire de services ainsi que sa clé publique qui servira lors de l'authentification d'un client.

Le prestataire de services est dès lors autorisé à charger ou décharger des services de la carte selon les choix du propriétaire.

En résumé:

- un prestataire de services non accrédité sera découvert par la carte lors de la première étape; la signature du certificat n'étant pas correcte.

- un prestataire de services non accrédité mais disposant du bon certificat sera découvert à la troisième étape grâce au nombre aléatoire.

- une carte non accréditée ou non accréditée mais disposant d'un bon certificat sera découverte à la deuxième étape grâce au nombre aléatoire.

3.3.3.2 Carte - Client

Le mécanisme d'authentification entre une carte et un client (Figure 3 - 8) est presque identique à celui présenté ci-dessus. Nous ne reviendrons donc pas sur les similitudes mais mettrons en évidence les différences.

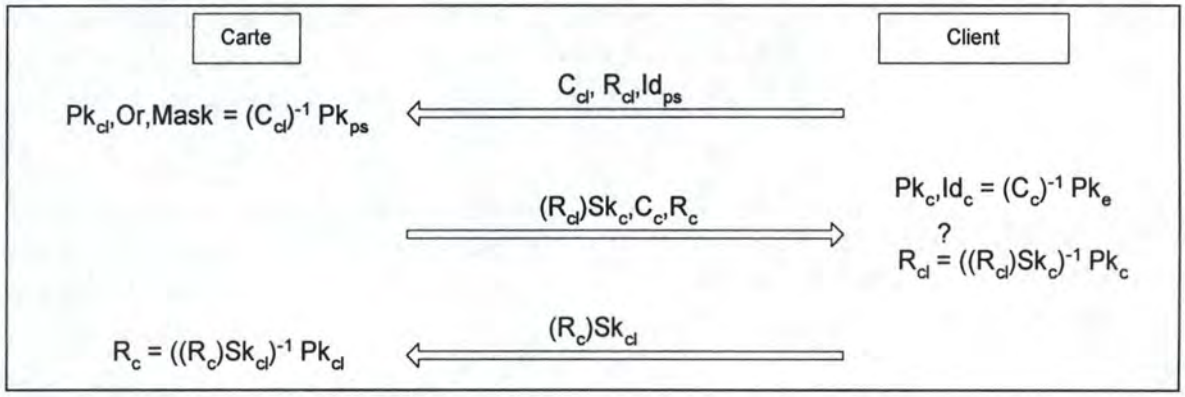


Figure 3 - 8 : Authentification d'un client.

Nous allons découper le protocole en trois phases, chacune d'elle correspondant à une flèche sur la Figure 3 - 8 montrant l'entièreté de l'authentification.

Client → Carte : C_{cl}, R_{cl}, Id_{ps}

La première étape est quelque peu différente dans cette authentification. En effet, le client doit envoyer, outre les informations permettant de se faire authentifier, l'identifiant du prestataire de services (Id_{ps}) qui l'a accrédité. La carte va pouvoir alors rechercher si ce prestataire de services s'est déjà fait authentifier. Si c'est le cas, elle dispose de la clé publique du prestataire de services. Dans le cas contraire le protocole est arrêté jusqu'à ce que ce prestataire s'authentifie auprès de la carte.

Le mécanisme d'authentification grâce à la vérification de la signature est identique à ce qui a été présenté au cours de l'authentification client - prestataire de services.

Carte → Client : $(R_{cl})Sk_c$, C_c, R_c

Le mécanisme d'authentification est identique à la deuxième étape de l'authentification client - prestataire de services.

Client → Carte : $(R_c)Sk_{cl}$

Cette étape est identique à la troisième étape de l'authentification client - prestataire de services.

Au terme de cette authentification mutuelle, le client est autorisé à exécuter des services de la carte selon bien entendu les droits dont il dispose. Ces droits sont transmis via le Mask et l'Object Reference et sont analysés par la carte lors d'une demande d'exécution.

Partie 2 : La Réalisation du protocole

Chapitre 4 : La plate-forme

Sommaire

Nous entrons par ce premier chapitre de la deuxième partie, dans la réalisation du protocole. Ce chapitre est consacré à la présentation, après un bref rappel des objectifs, aux outils utilisés et à la méthodologie suivie pour réaliser l'implémentation du protocole d'accès des partenaires à la carte.

4.1 Rappel des objectifs

L'objectif de ce mémoire est d'implémenter le protocole de sécurité des accès des partenaires (prestataires de services et clients) à la carte. Les objectifs sont dans un premier temps, de montrer la faisabilité du protocole mettant en présence une carte, un prestataire de services et un client. Et dans un second temps, de montrer que le protocole fonctionne toujours lorsque nous l'utilisons avec plusieurs partenaires.

4.2 Les outils utilisés

Le protocole repose sur un mécanisme d'authentification basé sur des certificats cryptographiques à clés publiques. Il nous fallait donc une carte à puce disposant d'un moteur cryptographique. En ce qui concerne les partenaires de la carte, nous avons décidé de les simuler à l'aide d'un PC.

La carte Combo n'existant pour l'instant que sur papier, il a fallu s'orienter vers une autre carte. C'est ainsi que nous avons opté pour la carte à clé publique GPK2000, produite par GEMPLUS, incluant un moteur cryptographique.

Malheureusement, les cartes GPK2000 n'étaient pas disponibles au cours de ce stage et GEMPLUS ne disposait pas d'autres cartes offrant des fonctions cryptographiques. Il a donc fallu se diriger vers une solution où les cartes, les clients et les prestataires de services seraient totalement simulés par un PC.

Du point de vue logiciel, nous avons utilisé le Resource Workshop du Visual C++ pour la création des boîtes de dialogue, fenêtres et menus et le C standard pour la programmation. Le moteur cryptographique a dû lui aussi être simulé grâce à la libraire *Cryptoki* spécifiée par le standard PKCS#11 que nous détaillerons dans le chapitre 5.

4.3 Méthodologie

Nous avons suivi une méthodologie que nous vous présentons à la Figure 4 - 1.

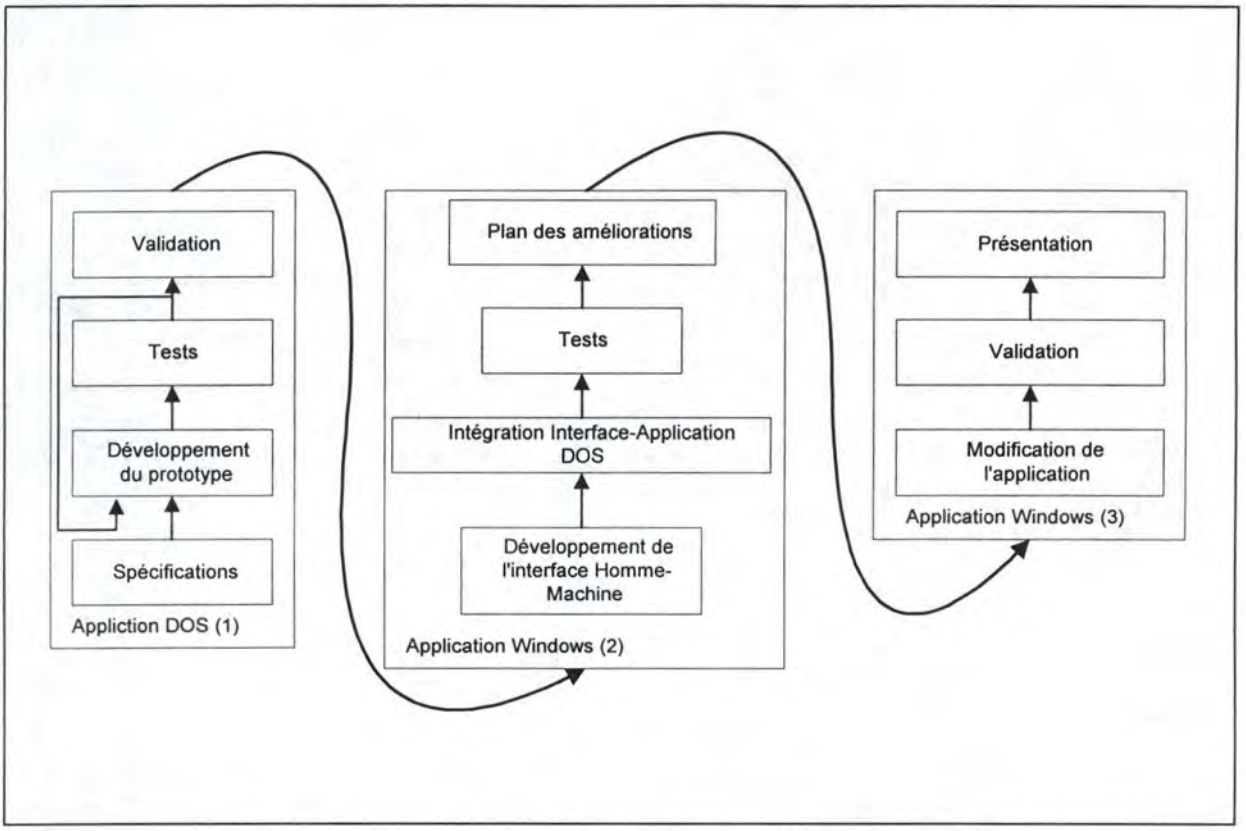


Figure 4 - 1 : Méthodologie de l'implémentation.

Ne sachant pas si nous pouvions arriver à mettre en oeuvre le protocole à l'aide de la librairie, nous avons d'abord opté pour l'implémentation d'un prototype fonctionnant sous DOS. Cette application fonctionnait alors avec une carte, un prestataire de services et un client. Au terme de celle-ci, nous disposions d'un outil nous permettant de tester la validité du protocole face à différentes attaques possibles (faux prestataire de services, fausse carte, faux client). Après quelques tests concluants, nous avons pu valider le protocole et nous concentrer alors sur la deuxième étape.

Cette deuxième étape consistait à porter l'application, développée en DOS, dans l'environnement Windows 95. Nous avons alors développé une interface constituée d'un menu et de boîtes de dialogue. Un exemple vous est présenté dans le dernier chapitre. Nous avons ensuite décidé de maintenir une séparation entre l'interface et le corps de l'application réalisant le protocole. Cette volonté est justifiée par le fait qu'il doit être possible de modifier une partie (interface ou corps du protocole) sans en affecter l'autre.

A la fin de cette étape, nous disposions d'une application tournant, sous Windows 95, avec un partenaire de chaque catégorie (une carte, un prestataire de services et un client).

Nous avons ensuite effectué quelques tests pour d'identifier les erreurs éventuelles, pour tester la résistance du protocole face à différents cas d'attaque (identiques à ceux testés sur l'application DOS) et pour mettre en évidence les améliorations possibles.

Nous avons ainsi pu identifier trois modifications :

- l'application devait pourvoir accueillir un nombre indéterminé de partenaires,
- l'application devait permettre à un client d'être accrédité par plusieurs prestataires de services,
- l'application devait permettre à un prestataire de services d'accréditer un client même s'il n'a pas encore chargé d'application dans une carte.

Ces modifications ont été apportées et nous sommes arrivés à une application tournant sous Windows 95 et pouvant mettre en oeuvre un nombre illimité de partenaires.

Chapitre 5 : Le standard PKCS#11

Sommaire

Ce chapitre a été rédigé sur base de deux documents :

- Bruton S. Kaliski Jr. - An Overview of the PKCS Standards - 1 Novembre 1993 [BKA,93],
- PKCS #11 : Cryptographic Token Interface Standard - RSA Laboratories - 28 Avril 1995 [RSA,95].

L'application développée mettant en oeuvre le protocole de sécurité utilise une librairie appelée *Cryptoki*. Cette librairie répond à des standards et dispose d'une série de fonctions que nous décrirons dans la section 5.4 de ce chapitre. Nous rattacherons ensuite cette librairie au protocole et identifierons les fonctions qu'il faudra utiliser pour le réaliser. Nous réécrirons, pour terminer, le protocole à l'aide de ces fonctions.

Pour des raisons de clarté et de compréhension, les différentes fonctions seront classées par des catégories.

5.1 Présentation générale

Vu que la cryptographie à clé publique commence à se répandre dans différentes applications, des standards d'interopérabilité doivent être mis en place.
Bien que les distributeurs soient d'accord sur les techniques à clé publique de base, la compatibilité entre les implémentations n'est en aucun cas garantie.

L'interopérabilité requiert par conséquent une acceptation et une adhésion à un format de standards pour le transfert des données. Le standard décrit dans ce chapitre fournit les bases pour l'interopérabilité. Nous appelons le standard décrit ici « Public-Key Cryptography Standards » ou PKCS.

PKCS décrit la syntaxe des messages de manière abstraite et donne des détails complets à propos des algorithmes. Cependant il ne spécifie pas comment les messages doivent être représentés. Donc, les implémentations de PKCS sont exemptes de messages échangés, de dépendance à un ensemble de caractères, de contraintes de tailles de tableaux, ...

Les standards PKCS ont été élaborés par les laboratoires RSA pour les développeurs de systèmes informatiques employant les technologies à clé publique, leurs intentions étant d'améliorer et de raffiner les standards en tenant compte des critiques et souhaits des développeurs dans le but de produire des standards qu'ils utiliseront tous.

5.2 Présentation du standard PKCS#11

PKCS#11 est la onzième version des standards développés par les laboratoires RSA. Ce standard spécifie une interface de programmation d'application (API) appelé Cryptoki (cryptographic token interface) réalisant des fonctions cryptographiques. Cette API permet d'isoler l'application des détails liés à l'appareil cryptographique. L'application ne doit pas changer d'interface lorsqu'elle est utilisée sur des appareils différents. Elle est alors portable. La manière dont cela est possible dépasse la portée de ce travail mais nous pouvons néanmoins dire que cela se réalise au travers de conventions pour le support de multiples types d'appareils.

Au départ, Cryptoki avait été conçue comme une simple interface entre l'application et tout type d'appareil cryptographique portable. Des standards d'interfaçage existaient déjà. Il restait donc à réaliser une API permettant de réaliser des commandes cryptographiques particulières. Le but a donc été de développer une API utilisable par toutes sortes d'appareils et réalisant des fonctions cryptographiques particulières.

5.3 Modèle général de PKCS#11

Le modèle (Figure 5 - 1) commence par une série d'applications dont le but est de réaliser certaines opérations reposant sur des fonctions cryptographiques. Il se termine avec des appareils cryptographiques sur lesquelles certaines opérations doivent être réalisées.

Cryptoki fournit l'interface pour ces appareils au travers d'un nombre de slots. Chaque slot (un lecteur physique), peut contenir un token (vue logique d'un appareil cryptographique). Un appareil cryptographique étant par exemple une carte à puce.

Grâce à Cryptoki, l'application n'a pas besoin de communiquer à l'aide des drivers de l'appareil, la librairie Cryptoki se chargeant des conversions nécessaires. Le lien entre une application et un ou plusieurs token se réalise au moyen d'un numéro de session.

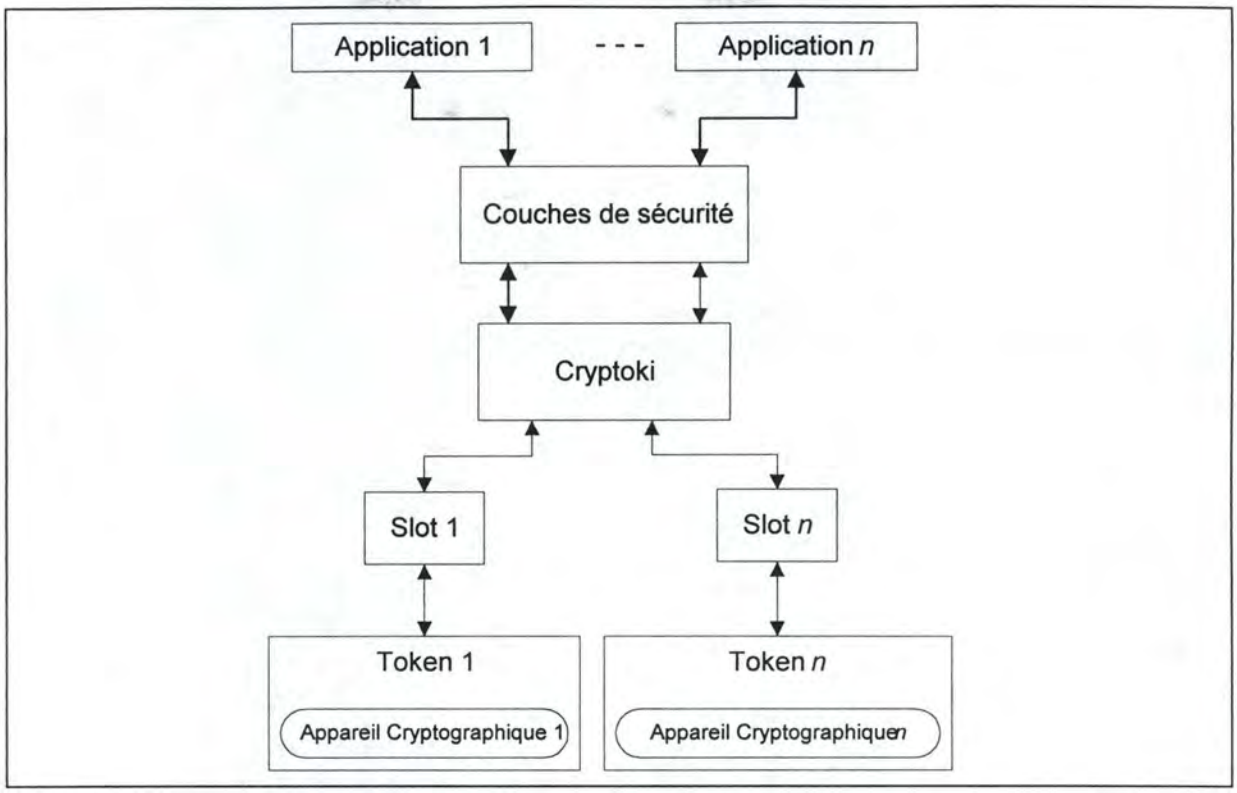


Figure 5 - 1 : Modèle général.

Un token est un appareil qui stocke des objets et réalise des fonctions cryptographiques. Cryptoki définit trois classes d'objet : les données, les certificats et les clés. Nous reviendrons plus en détail sur ces objets au cours de la section suivante consacrée à la description de l'API.

5.4 Description de l'API

Nous allons maintenant décrire les fonctions de l'API. Pour des raisons de clarté et de compréhension, nous n'indiquerons pas les types des paramètres de ces fonctions. Nous avons donc choisi de vous les présenter en terme de leur entête et des paramètres reçus.

Chaque fonction, au terme de son exécution, renvoie un numéro permettant de déterminer si celle-ci s'est bien déroulée ou pas. Si l'exécution s'est mal déroulée, le numéro permettra d'identifier l'erreur.

Les descriptifs qui seront présentés au cours des sous-sections ci-dessous contiennent une liste d'attributs nécessaires pour que les fonctions qui les utilisent s'effectuent correctement. D'autres attributs facultatifs peuvent y être ajoutés. Pour des raisons de clarté et afin d'éviter de devoir entrer dans des explications compliquées, nous avons préféré ne pas en parler.

5.4.1 Définitions

Au cours des sous-sections présentées ci-dessous, un certain nombre de concepts et abréviations vont être abordés. Afin de faciliter leur compréhension, nous allons les définir [BSH,97] [BKA,93] [JRA,97] :

- DSA
  DSA est présenté à l'annexe A2.

- Mode CBC
  Le mode CBC (Figure 5 - 2) (chaînage de blocs chiffrés) est obtenu en combinant, par ou-exclusif, le bloc chiffré sortant au bloc en clair suivant pour former le bloc entrant suivant. Il est nécessaire, pour démarrer ce mode de fonctionnement, de choisir une valeur initiale. Le choix de cette valeur permet de donner des chiffrés différents à des textes en clairs identiques.

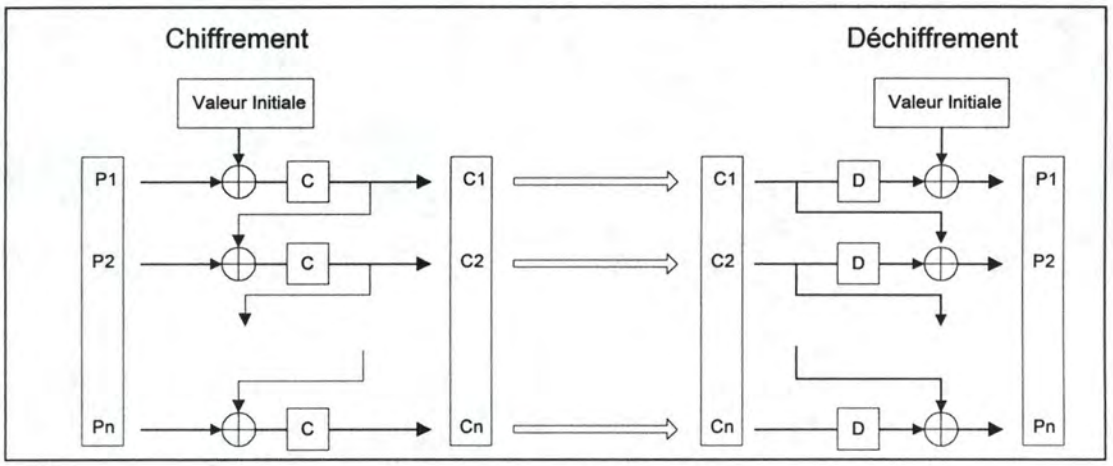


Figure 5 - 2 : Mode CBC.

- Mode ECB (Electronic CodeBook)
  Par le mode ECB (Figure 5 - 3), le message à chiffrer est découpé en blocs et chacun de ceux-ci est chiffré séparément. Le message chiffré est ainsi constitué des différents blocs chiffrés.

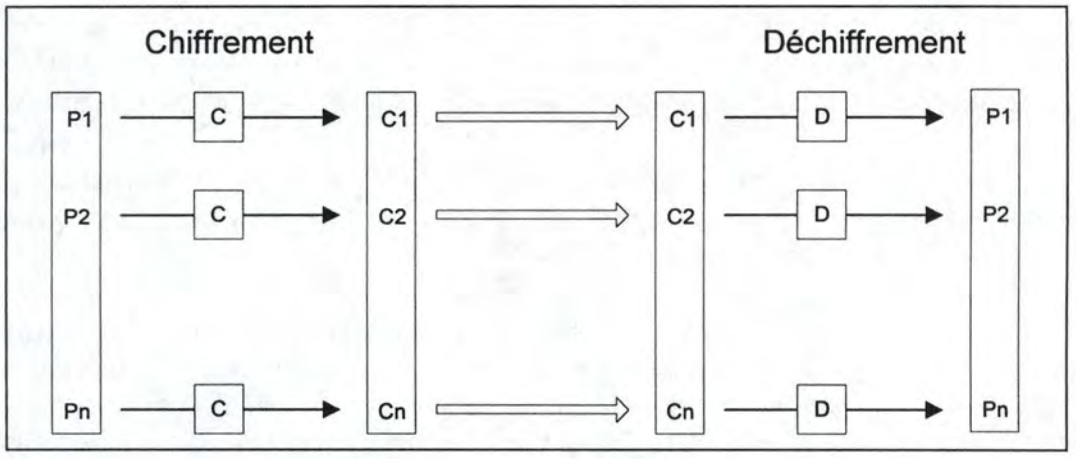


Figure 5 - 3 : Mode ECB.

- DES
  Data Encryption Standard. Il est présenté plus en détail à l'annexe A0.

- Double DES
  Le chiffrement, à l'aide de l'algorithme DES, s'effectue deux fois. On chiffre le message avec une première clé. Le résultat est alors à nouveau chiffré avec une seconde clé.

- Triple DES
  Le chiffrement, à l'aide de l'algorithme DES, s'effectue trois fois. On chiffre le message avec une première clé. Le résultat est alors à nouveau chiffré avec une seconde clé. Le résultat du nouveau chiffrement est chiffré avec la première clé.

- DES-MAC
  Avant d'expliquer ce que signifie le DES-MAC, précisons ce que signifie MAC. Le MAC est un mécanisme employé lorsqu'on ne souhaite pas envoyer un message entièrement chiffré. Il consiste à calculer un paramètre caractéristique du texte en clair et à envoyer le texte en clair suivi de ce paramètre appelé Message Authentication Code (MAC).

  Le DES-MAC (Figure 5 - 4) consiste à calculer le MAC à l'aide d'un algorithme DES.

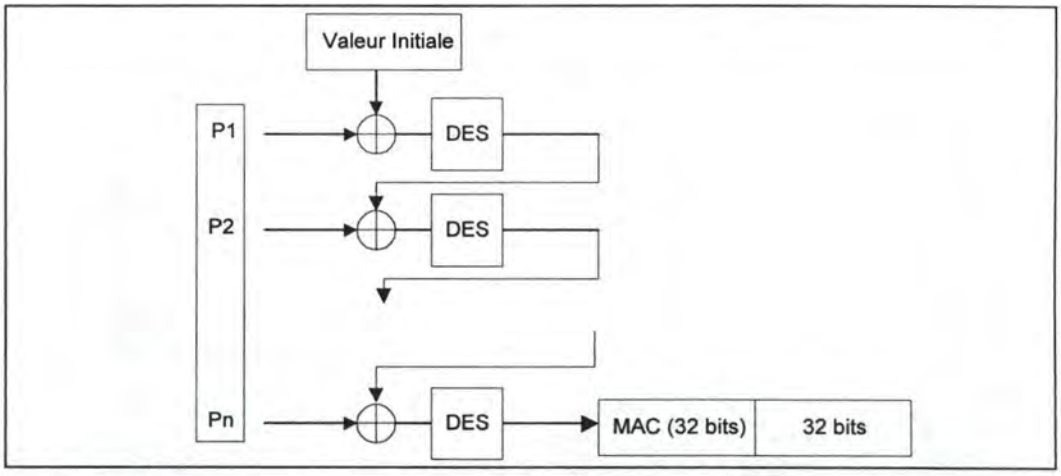


Figure 5 - 4 : Calcul d'un MAC à partir du DES.

- DES-CBC
  DES-CBC est un mécanisme par lequel le chiffrement des blocs (comme expliqué ci-dessus : Mode CBC) se fait au moyen de l'algorithme DES.

- DES-ECB
  DES-ECB est un mécanisme par lequel le chiffrement des blocs (comme expliqué ci-dessus : Mode ECB) se fait au moyen de l'algorithme DES.

- RSA
  RSA est présenté en détail à l'annexe A1.

- X.509 RSA
  X.509 RSA est un mécanisme de chiffrement basé sur le cryptosystème à clé publique RSA.

- PKCS#1 RSA
  PKCS#1 RSA décrit un mécanisme de chiffrement et signature de données utilisant un système cryptographique à clé publique.

- Diffie-Hellman
  Diffie-Hellman est présenté à l'annexe A3.

- PKCS#3 Diffie-Hellman
  PKCS#3 Diffie-Hellman définit un mécanisme d'implémentation de clés Diffie-Hellman dans lequel deux parties, sans accord préalable, peuvent être d'accord sur le choix d'une clé secrète. La clé secrète étant utilisée pour le chiffrement des communications à venir.

- RC2
  RC2 est un algorithme de chiffrement à clé secrète de taille variable. Il permet de chiffrer des blocs d'une longueur de 64 bits à l'aide d'une clé dont la taille peut varier de 0 bytes à la longueur d'un mot accepté par un ordinateur.

- RC2-CBC
  RC2-CBC est un mécanisme par lequel le chiffrement des blocs (comme expliqué ci-dessus : Mode CBC) se fait au moyen de l'algorithme RC2.

- RC2-ECB
  RC2-ECB est un mécanisme par lequel le chiffrement des blocs (comme expliqué ci-dessus : Mode ECB) se fait au moyen de l'algorithme RC2.

- RC2-MAC
  Le RC2-MAC consiste à calculer le MAC à l'aide d'un algorithme RC2.

- RC4
  RC4 est un algorithme de chiffrement en continu, à clé secrète de taille variable. Le chiffrement s'effectue à l'aide d'une clé dont la taille peut varier de 0 bytes à la longueur d'un mot accepté par un ordinateur.

- ISO/IEC 9796 RSA
  ISO/IEC 9796 RSA définit un standard de signature digitale RSA.

5.4.2 Les objets et leur descriptif

Nous allons présenter dans cette sous-section différents descriptifs comprenant des attributs. A chaque attribut correspond un couple (valeur, taille de la valeur). Pour des raisons de simplicité, nous n'indiquerons pas la seconde composante du couple.

Les données
Ces objets contiennent de l'information définie par l'application.

Le descriptif d'un tel objet comprend :

- le type d'objet (dans ce cas le type est données)
- le label de l'objet,
- le caractère de l'objet : privé ou public,
- le label de l'application qui gère cet objet,
- les données.

Les certificats
Ces objets contiennent des certificats.

Le descriptif d'un tel objet comprend :

- le type d'objet (dans ce cas le type est certificat)
- le label de l'objet,
- le caractère de l'objet : privé ou public,
- le type de certificat : X.509 ou défini par le vendeur,
- la valeur du certificat.

Les clés publiques
Cryptoki reconnaît trois types de clé publique :

1. Clés publiques RSA
   Le descriptif de cet objet comprend :
   - le type d'objet (dans ce cas le type est clé),
   - le type de clé (dans ce cas le type est public-key RSA),
   - le label de la clé,
   - l'identifiant de la clé,
   - le Modulo n,
   - la longueur (en bits) de la clé que l'on désire créer,
   - l'exposant public e.

   L'algorithme permettant de générer les clés se trouve à l'annexe A1.

2. Clés publiques DSA
   Le descriptif de cet objet comprend :
   - le type d'objet (dans ce cas le type est clé),
   - le type de clé (dans ce cas le type est public-key DSA),
   - le label de la clé,
   - l'identifiant de la clé,
   - le nombre premier p,
   - q, un nombre premier à p-1,
   - la base g,
   - la valeur publique y.

3. Clés publiques Diffie-Hellman

Le descriptif de cet objet comprend :

- le type d'objet (dans ce cas le type est clé),
- le type de clé (dans ce cas le type est public-key Diffie-Hellman),
- le label de la clé,
- l'identifiant de la clé,
- le nombre premier *p*,
- la base *g*,
- la valeur publique *y*.

Les clés secrètes

Cryptoki reconnaît trois types de clé secrète :

1. Clés secrètes RSA

Le descriptif de cet objet comprend :

- le type d'objet (dans ce cas le type est clé),
- le type de clé (dans ce cas le type est private-key RSA),
- le label de la clé,
- l'identifiant de la clé,
- le Modulo *n*,
- l'exposant public *e*,
- l'exposant privé *d*,
- les deux nombres premiers *p* et *q*,
- l'exposant privé *d* modulo *p*-1,
- l'exposant privé *d* modulo *q*-1,
- le coefficient q^{-1} modulo *p*.

2. Clés secrètes DSA

Le descriptif de cet objet comprend :

- le type d'objet (dans ce cas le type est clé),
- le type de clé (dans ce cas le type est private-key DSA),
- le label de la clé,
- l'identifiant de la clé,
- le nombre premier *p*,
- *q*, un nombre premier à *p-1*,
- la base *g*,
- la valeur secrète *x*.

3. Clés secrètes Diffie-Hellman

Le descriptif de cet objet comprend :

- le type d'objet (dans ce cas le type est clé),
- le type de clé (dans ce cas le type est private-key Diffie-Hellman),
- le label de la clé,
- l'identifiant de la clé,
- le nombre premier *p*,
- la base *g*,
- la valeur secrète *x*,
- la longueur (en bits) de la valeur secrète *x*.

5.4.3 Fonctions générales

C_Initialize
Cette fonction initialise la librairie Cryptoki (ex. : initialisation des buffers en mémoire). C'est la première instruction que toute application doit appeler. Elle ne reçoit aucun paramètre.

C_GetInfo
Cette fonction retourne des informations générales à propos de Cryptoki. Elle reçoit comme paramètre un pointeur sur une zone qui recevra les informations.

C_GetSlotList
Cette fonction permet de recevoir la liste des slots du système.

Elle reçoit comme paramètres :
- la zone de réception de la liste des slots (l'identifiant des slots),
- la zone de réception du nombre de slots.

Cette fonction doit être appelée deux fois. La première fois en donnant une valeur nulle au premier paramètre. La fonction retourne alors le nombre de slots présents dans le système. La seconde fois en ne donnant aucune valeur aux paramètres. La fonction retourne alors la liste des slots présents dans le système.

C_GetSlotInfo
Cette fonction retourne de l'information à propos d'un slot particulier du système.

Elle reçoit comme paramètres :
- l'identifiant du slot,
- la zone de réception de l'information.

L'information renvoyée comprend :
- la description du slot (le type d'interface entre l'appareil et l'ordinateur)
- l'identifiant du fabricant du slot
- la présence ou non d'un token dans le slot
- le type de slot (software ou hardware)

C_GetTokenInfo
Cette fonction retourne de l'information à propos d'un token particulier du système.

Elle reçoit comme paramètres :
- l'identifiant du slot contenant le token,
- pointeur sur la zone de réception de l'information.

L'information renvoyée comprend :
- le nom de l'application défini lors de l'initialisation d'un token,
- l'identifiant du fabricant de l'appareil,
- le modèle de l'appareil,
- le numéro de série de l'appareil,

- le nombre de sessions qui peuvent être ouvertes avec le token,
- le nombre de sessions qui sont actuellement ouvertes avec le token,
- le nombre de sessions lecture/écriture qui peuvent être ouvertes avec le token,
- le nombre de sessions lecture/écriture qui sont ouvertes avec le token,
- la longueur maximum du numéro d'identification personnel (PIN),
- la longueur minimum du numéro d'identification personnel (PIN),
- la quantité de mémoire utilisée par les objets publics,
- la quantité de mémoire libre pour les objets publics,
- la quantité de mémoire utilisée par les objets privés,
- la quantité de mémoire libre pour les objets privés,
- la présence d'un générateur de nombre aléatoire dans le token,
- la protection en écriture sur le token,
- l'obligation de se connecter pour effectuer des fonctions cryptographiques,
- l'initialisation du numéro d'identification personnel a été effectuée.

`C_GetMechanismList`
Cette fonction retourne la liste des mécanismes supportés par un token.

Elle reçoit comme paramètres :

- l'identifiant du slot contenant le token,
- la zone de réception de la liste des types de mécanismes,
- la zone de réception du nombre de mécanismes.

Cette fonction doit être appelée deux fois. La première fois en donnant une valeur nulle au second paramètre. La fonction retourne alors le nombre de mécanismes supportés par le token. La seconde fois en ne donnant aucune valeur aux paramètres. La fonction retourne alors la liste des mécanismes.

`C_GetMechanismInfo`
Cette fonction retourne de l'information à propos d'un mécanisme particulier supporté par un token.

Elle reçoit comme paramètres :

- l'identifiant du slot contenant le token,
- le type de mécanisme pour lequel on souhaite obtenir de l'information,
- la zone de réception de l'information.

L'information renvoyée comprend :

- la taille minimum des clés utilisées par ce mécanisme,
- la taille maximum des clés utilisées par ce mécanisme.

`C_InitToken`
Cette fonction initialise un token.

Elle reçoit comme paramètres :

- l'identifiant du slot contenant le token,
- le numéro d'identification personnel du « superviseur »,
- la longueur de ce numéro,

- le nom que l'on souhaite donner au token.

Un utilisateur normal n'aura accès à un token qu'une fois que le superviseur lui aura attribué un numéro d'identification personnel.

C_InitPin
Cette fonction initialise le numéro d'identification personnel de l'utilisateur normal.
Elle reçoit comme paramètres :

- le numéro d'identification personnel de l'utilisateur normal,
- la longueur de ce numéro,
- le numéro de session.

C_SetPin
Cette fonction permet de modifier le numéro d'identification personnel de l'utilisateur connecté.

Elle reçoit comme paramètres :

- l'ancien numéro d'identification personnel de l'utilisateur,
- la longueur de ce numéro,
- le nouveau numéro d'identification personnel de l'utilisateur,
- la longueur de ce numéro,
- le numéro de session.

C_OpenSession
Cette fonction permet d'ouvrir une session entre une application et un token.

Elle reçoit comme paramètres :

- l'identifiant d'un slot,
- le type de session (Read Only, Read/Write,...),
- la zone de réception du numéro de session.

C_CloseSession
Cette fonction ferme une session. Elle reçoit comme paramètres le numéro de la session à fermer. Les objets créés durant la session sont alors détruits.

C_CloseAllSessions
Cette fonction ferme toutes les sessions d'un token donné. Elle reçoit comme paramètres l'identifiant du slot contenant le token dont il faut fermer les sessions. Les objets créés durant la session sont alors détruits.

C_GetSessionInfo
Cette fonction permet d'obtenir de l'information sur une session.

Elle reçoit comme paramètres :

- le numéro de la session,
- la zone de réception de l'information.

L'information renvoyée comprend :

- le numéro de version de Cryptoki,
- l'identifiant du fabricant de Cryptoki.

C_Login
Cette fonction connecte un utilisateur à un token.
Elle reçoit comme paramètres :

- le numéro de la session,
- le type d'utilisateur (superviseur ou utilisateur normal),
- le numéro d'identification personnel de l'utilisateur.

C_Logout
Cette fonction déconnecte un utilisateur d'un token. Elle reçoit comme paramètre le numéro de la session.

5.4.4 Gestion des objets

C_CreateObject
Cette fonction permet de créer un nouvel objet. Uniquement les objets publics pourront être créés si aucun utilisateur n'est connecté.

Elle reçoit comme paramètres :

- le numéro de la session,
- le descriptif de l'objet,
- le nombre d'attribut contenu dans le descriptif,
- la zone de réception du numéro de l'objet.

C_CopyObject
Cette fonction permet de copier un objet.

Elle reçoit comme paramètres :

- le numéro de la session,
- le numéro de l'objet à copier,
- le descriptif du nouvel objet,
- le nombre d'attribut contenu dans le descriptif de l'objet,
- la zone de réception du numéro du nouvel objet.

C_DestroyObject
Cette fonction permet de détruire un objet.

Elle reçoit comme paramètres :

- le numéro de la session,
- le numéro de l'objet à détruire.

Uniquement les objets publics pourront être détruits si aucun utilisateur n'est connecté.

C_GetObjectSize
Cette fonction renvoie la taille (en bytes) d'un objet.

Elle reçoit comme paramètres :

- le numéro de la session,
- le numéro de l'objet dont on souhaite connaître la taille,
- la zone de réception de la taille.

C_GetAttributeValue

Cette fonction permet d'obtenir la valeur d'un ou plusieurs attributs contenus dans le descriptif d'un objet à condition qu'il ne soit pas considéré comme objet « critique ».

Elle reçoit comme paramètres :

- le numéro de la session,
- le numéro de l'objet dont on souhaite connaître la valeur d'un ou plusieurs attributs,
- le descriptif contenant les attributs dont on souhaite connaître la valeur,
- le nombre d'attributs de ce descriptif.

C_SetAttributeValue

Cette fonction permet de modifier la valeur d'un ou plusieurs attributs contenus dans le descriptif d'un objet.

Elle reçoit comme paramètres :

- le numéro de la session,
- le numéro de l'objet dont on souhaite modifier la valeur d'un ou plusieurs attributs,
- le descriptif contenant les attributs que l'on souhaite modifier ainsi que leurs nouvelles valeurs,
- le nombre d'attributs de ce descriptif.

C_FindObjectsInit

Cette fonction initialise une recherche de token et d'objets qui répondent à un descriptif donné. Après avoir appelé cette fonction, l'application peut appeler la fonction `C_FindObjects` autant de fois qu'elle le désire.

Elle reçoit comme paramètres :

- le numéro de la session,
- le descriptif contenant la valeur des attributs auquel l'objet recherché devra se conformer,
- le nombre d'attributs de ce descriptif.

C_FindObjects

Cette fonction continue la recherche de token ou d'objets qui correspondent à un descriptif donné. Cette fonction ne peut être appelée que si la fonction `C_FindObjectsInit` a été appelée au moins une fois.

Elle reçoit comme paramètres :

- le numéro de la session,
- la zone de réception de la liste des numéros des objets répondant au descriptif

- le nombre maximum d'objets que l'on souhaite recevoir,
- le numéro du dernier objet trouvé.

Si aucun objet n'est trouvé, le numéro du dernier objet trouvé est .

5.4.5 Chiffrement et Déchiffrement

C_EncryptInit

Cette fonction a pour but d'initialiser une opération de chiffrement. L'opération de chiffrement n'est « active » que jusqu'au moment où vous appelez *C_Encrypt*, *C_EncryptUpdate* ou *C_EncryptFinal*. Vous devez alors rappeler la fonction *C_EncryptInit* pour réinitialiser l'opération de chiffrement.

Vous ne pouvez réaliser les fonctions *C_Encrypt*, *C_EncryptUpdate*, *C_EncryptFinal* qu'après avoir appelé cette fonction.

Cette fonction ne peut pas initialiser une nouvelle opération de chiffrement si une autre est toujours active.

Elle reçoit comme paramètres :

- un numéro de session,
- le type de mécanisme de chiffrement (Tableau 5 - 1),
- le numéro de la clé de chiffrement qui représente le numéro de l'objet contenant cette clé.

Mécanisme	Type de clé
PKCS#1 RSA (1)	Publique RSA
X.509 RSA (1)	Publique RSA
RC2 (mode ECB et CBC)	RC2
RC4	RC4
DES (mode ECB et CBC)	DES
triple-DES (mode ECB et CBC)	double ou triple DES

Tableau 5 - 1 : Mécanismes de chiffrement.

(1) Uniquement pour des chiffrements en un seul bloc.

C_Encrypt

Cette fonction chiffre des données en un seul bloc. L'opération de chiffrement doit avoir été initialisée à l'aide de la fonction *C_EncryptInit* avant tout appel à la fonction *C_Encrypt*.

Elle reçoit comme paramètres :

- un numéro de session,
- un pointeur sur la donnée à chiffrer,
- la longueur (en bytes) de la donnée à chiffrer,
- un pointeur sur la zone qui contiendra la donnée chiffrée,
- un pointeur sur la zone qui recevra la longueur (en bytes) de la donnée chiffrée.

Selon le mécanisme de chiffrement choisi, les données doivent avoir une longueur dépendante de la longueur *k* de la clé de chiffrement (Tableau 5 - 2).

Mécanisme	Type de clé	Longueur de la donnée (bytes)	Longueur du résultat (bytes)
PKCS#1 RSA	Publique RSA	<=k-11	k
X.509 RSA	Publique RSA	<= k	k
RC2-ECB	RC2	multiple de 8	même longueur que la donnée
RC2-CBC	RC2	multiple de 8	même longueur que la donnée
RC4	RC4	quelconque	même longueur que la donnée
DES-ECB	DES	multiple de 8	même longueur que la donnée
DES-CBC	DES	multiple de 8	même longueur que la donnée
Triple DES-ECB	2-DES ou 3-DES	multiple de 8	même longueur que la donnée
Triple DES-CBC	2-DES ou 3-DES	multiple de 8	même longueur que la donnée

Tableau 5 - 2 : Contraintes liées aux mécanismes de chiffrement.

`C_EncryptUpdate`

Cette fonction continue une opération de chiffrement en plusieurs parties. L'opération de chiffrement doit avoir été initialisée à l'aide de la fonction *`C_EncryptInit`* avant tout appel à la fonction *`C_EncryptUpdate`*.

Elle reçoit comme paramètres :

- un numéro de session,
- un pointeur sur la donnée à chiffrer,
- la longueur (en bytes) de la donnée à chiffrer,
- un pointeur sur la zone qui contiendra la donnée chiffrée,
- un pointeur sur la zone qui recevra la longueur (en bytes) de la donnée chiffrée.

Selon le mécanisme de chiffrement choisi, les données doivent avoir une longueur dépendante de la longueur *k* de la clé de chiffrement (Tableau 5 - 3).

Mécanisme	Type de clé	Longueur de la donnée (bytes)	Longueur du résultat (bytes)
RC2-ECB	RC2	multiple de 8	même longueur que la donnée
RC2-CBC	RC2	multiple de 8	même longueur que la donnée
RC4	RC4	quelconque	même longueur que la donnée
DES-ECB	DES	multiple de 8	même longueur que la donnée
DES-CBC	DES	multiple de 8	même longueur que la donnée
Triple DES-ECB	2-DES ou 3-DES	multiple de 8	même longueur que la donnée
Triple DES-CBC	2-DES ou 3-DES	multiple de 8	même longueur que la donnée

Tableau 5 - 3 : Contraintes liées aux mécanismes de chiffrement.

`C_EncryptFinal`

Cette fonction termine une opération de chiffrement en plusieurs parties. L'opération de chiffrement doit avoir été initialisée à l'aide de la fonction *`C_EncryptInit`* avant tout appel à la fonction *`C_EncryptUpdate`*.

Elle reçoit comme paramètres :

- un numéro de session,
- un pointeur sur la zone qui contiendra la donnée chiffrée,
- un pointeur sur la zone qui recevra la longueur (en bytes) de la donnée chiffrée.

Les contraintes sur la longueur des données sont identiques à celles de la fonction précédente (Tableau 5 - 3).

`C_DecryptInit`

Cette fonction a pour but d'initialiser une opération de déchiffrement. L'opération de déchiffrement n'est « active » que jusqu'au moment où vous appelez `C_Decrypt`, `C_DecryptFinal` ou `C_DecryptUpdate`. Vous devez alors rappeler la fonction `C_EncryptInit` pour réinitialiser l'opération de déchiffrement.

Vous ne pouvez réaliser les fonctions `C_Decrypt`, `C_DecryptUpdate`, `C_EncryptFinal` qu'après avoir appelé cette fonction.

Cette fonction ne peut pas initialiser une nouvelle opération de déchiffrement si une autre est toujours active.

Elle reçoit comme paramètres :

- un numéro de session,
- le type de mécanisme de déchiffrement (Tableau 5 - 4),
- le numéro de la clé de déchiffrement qui représente le numéro de l'objet contenant cette clé.

Mécanisme	Type de clé
PKCS#1 RSA (1)	Publique RSA
ISO/IEC 9796 RSA (1)	Publique RSA
X.509 RSA (1)	Publique RSA
DSA (1)	Privée DSA
RC2-MAC	RC2
DES-MAC	DES
triple-DES (mode ECB et CBC)	double ou triple DES

Tableau 5 - 4 : Mécanismes de déchiffrement.

(1) Uniquement pour des déchiffrements en un seul bloc.

`C_Decrypt`

Cette fonction déchiffre des données en un seul bloc. L'opération de déchiffrement doit avoir été initialisée à l'aide de la fonction `C_DecryptInit` avant tout appel à la fonction `C_Decrypt`.

Elle reçoit comme paramètres :

- un numéro de session,
- un pointeur sur la donnée à déchiffrer,
- la longueur (en bytes) de la donnée à déchiffrer,
- un pointeur sur la zone qui contiendra la donnée déchiffrée,

- un pointeur sur la zone qui recevra la longueur (en bytes) de la donnée déchiffrée.

Selon le mécanisme de déchiffrement choisi, les données doivent avoir une longueur dépendante de la longueur *k* de la clé de déchiffrement (Tableau 5 - 5).

Mécanisme	Type de clé	Longueur de la donnée (bytes)	Longueur du résultat (bytes)
PKCS#1 RSA	Privée RSA	k	<= k-11
X.509 RSA	Privée RSA	k	<= k
RC2-ECB	RC2	multiple de 8	même longueur que la donnée
RC2-CBC	RC2	multiple de 8	même longueur que la donnée
RC4	RC4	quelconque	même longueur que la donnée
DES-ECB	DES	multiple de 8	même longueur que la donnée
DES-CBC	DES	multiple de 8	même longueur que la donnée
Triple DES-ECB	2-DES ou 3-DES	multiple de 8	même longueur que la donnée
Triple DES-CBC	2-DES ou 3-DES	multiple de 8	même longueur que la donnée

Tableau 5 - 5 : Contraintes liées aux mécanismes de déchiffrement.

`C_DecryptUpdate`

Cette fonction continue une opération de déchiffrement en plusieurs parties. L'opération de déchiffrement doit avoir été initialisée à l'aide de la fonction `C_DecryptInit` avant tout appel à la fonction `C_DecryptUpdate`.

Elle reçoit comme paramètres :

- un numéro de session,
- un pointeur sur la donnée à déchiffrer,
- la longueur (en bytes) de la donnée à déchiffrer,
- un pointeur sur la zone qui contiendra la donnée déchiffrée,
- un pointeur sur la zone qui recevra la longueur (en bytes) de la donnée déchiffrée.

Selon le mécanisme de déchiffrement choisi, les données doivent avoir une longueur dépendante de la longueur *k* de la clé de déchiffrement (Tableau 5 - 6).

Mécanisme	Type de clé	Longueur de la donnée (bytes)	Longueur du résultat (bytes)
RC2-ECB	RC2	multiple de 8	même longueur que la donnée
RC2-CBC	RC2	multiple de 8	même longueur que la donnée
RC4	RC4	quelconque	même longueur que la donnée
DES-ECB	DES	multiple de 8	même longueur que la donnée
DES-CBC	DES	multiple de 8	même longueur que la donnée
Triple DES-ECB	2-DES ou 3-DES	multiple de 8	même longueur que la donnée
Triple DES-CBC	2-DES ou 3-DES	multiple de 8	même longueur que la donnée

Tableau 5 - 6 : Contraintes liées aux mécanismes de déchiffrement.

`C_DecryptFinal`

Cette fonction termine une opération de déchiffrement en plusieurs parties. L'opération de déchiffrement doit avoir été initialisée à l'aide de la fonction `C_DecryptInit` avant tout appel à la fonction `C_DecryptUpdate`.

Elle reçoit comme paramètres :

- un numéro de session,
- un pointeur sur la zone qui contiendra la donnée déchiffrée,
- un pointeur sur la zone qui recevra la longueur (en bytes) de la donnée déchiffrée.

Les contraintes sur la longueur des données sont identiques à celles de la fonction précédente (Tableau 5 - 6).

5.4.6 Signature et Vérification

C_SignInit

Cette fonction initialise l'opération de signature. La signature étant un appendice de la donnée.

L'opération de signature n'est « active » que jusqu'au moment où vous appelez *C_Sign*, *C_SignUpdate* ou *C_SignFinal*. Vous devez alors rappeler la fonction *C_SignInit* pour réinitialiser l'opération de signature.

Vous ne pouvez réaliser les fonctions *C_Sign*, *C_SignUpdate*, *C_SignFinal* qu'après avoir appelé cette fonction.

Cette fonction ne peut pas initialiser une nouvelle opération de signature si une autre est toujours active.

Elle reçoit comme paramètres :

- un numéro de session,
- le type de mécanisme de signature (Tableau 5 - 7),
- le numéro de la clé de signature qui représente le numéro de l'objet contenant cette clé.

Mécanisme	Type de clé
PKCS#1 RSA (1)	Privée RSA
ISO/IEC 9796 RSA (1)	Prviée RSA
X.509 RSA (1)	Privée RSA
DSA (1)	Privée DSA
RC2-MAC	RC2
DES-MAC	DES
triple-DES (mode ECB et CBC)	double ou triple DES

Tableau 5 - 7 : Mécanismes de signature.

(1) Uniquement pour des signatures en un seul bloc.

C_Sign

Cette fonction signe des données en un seul bloc; la signature étant un appendice des données. L'opération de signature doit avoir été initialisée à l'aide de la fonction *C_SignInit* avant tout appel à la fonction *C_Sign*.

Elle reçoit comme paramètres :

- un numéro de session,
- un pointeur sur la donnée à signer,
- la longueur (en bytes) de la donnée à signer,
- un pointeur sur la zone qui contiendra la signature,
- un pointeur sur la zone qui recevra la longueur (en bytes) de la signature.

Selon le mécanisme de signature choisi, les données doivent avoir une longueur dépendante de la longueur *k* de la clé de signature (Tableau 5 - 8).

Mécanisme	Type de clé	Longueur de la donnée (bytes)	Longueur de la signature (bytes)
PKCS#1 RSA	Privée RSA	<= k-11	k
ISO/IEC 9796 RSA	Privée RSA	<= val. inf. (k/2)	k
X.509 RSA	Privée RSA	<= k	k
DSA	Privée DSA	20	40
RC2-MAC	RC2	quelconque	4
DES-MAC	DES	quelconque	4
Triple DES	2-DES ou 3-DES	quelconque	4

Tableau 5 - 8 : Contraintes liées aux mécanismes de signature.

`C_SignUpdate`

Cette fonction continue une opération de signature en plusieurs parties. L'opération de signature doit avoir été initialisée à l'aide de la fonction `C_SignInit` avant tout appel à la fonction `C_SignUpdate`.

Elle reçoit comme paramètres :

- un numéro de session,
- un pointeur sur la donnée à signer,
- la longueur (en bytes) de la donnée à signer.

Selon le mécanisme de signature choisi, les données doivent avoir une longueur dépendante de la longueur *k* de la clé de signature (Tableau 5 - 9).

Mécanisme	Type de clé	Longueur de la donnée (bytes)	Longueur de la signature (bytes)
RC2-MAC	RC2	quelconque	4
DES-MAC	DES	quelconque	4
Triple DES	2-DES ou 3-DES	quelconque	4

Tableau 5 - 9 : Contraintes liées aux mécanismes de signature et vérification.

`C_SignFinal`

Cette fonction termine une opération de signature en plusieurs parties. L'opération de signature doit avoir été initialisée à l'aide de la fonction `C_SignInit` avant tout appel à la fonction `C_SignFinal`.

Elle reçoit comme paramètres :

- un numéro de session,
- un pointeur sur la zone qui contiendra la signature,
- un pointeur sur la zone qui recevra la longueur (en bytes) de la signature.

Les contraintes sur la longueur des données sont identiques à celles de la fonction précédente (Tableau 5 - 9).

`C_SignRecoverInit`

Cette fonction initialise une opération de signature; les données peuvent être retrouvées à partir de la signature.

L'opération de signature n'est « active » que jusqu'au moment où vous appelez `C_SignRecover`. Vous devez alors rappeler la fonction `C_SignRecoverInit` pour réinitialiser l'opération de signature.

Vous ne pouvez réaliser les fonctions `C_SignRecover` qu'après avoir appelé cette fonction.

Cette fonction ne peut pas initialiser une nouvelle opération de signature si une autre est toujours active.

Elle reçoit comme paramètres :

- un numéro de session,
- le type de mécanisme de signature (Tableau 5 - 10),
- le numéro de la clé de signature qui représente le numéro de l'objet contenant cette clé.

Mécanisme	Type de clé
PKCS#1 RSA	Privée RSA
ISO/IEC 9796 RSA	Privée RSA
X.509 RSA	Privée RSA

Tableau 5 - 10 : Mécanismes de signature.

`C_SignRecover`

Cette fonction signe des données en une seule opération; les données peuvent être retrouvées à partir de la signature.

L'opération de signature doit avoir été initialisée à l'aide de la fonction `C_SignRecoverInit` avant tout appel à la fonction `C_SignRecover`.

Elle reçoit comme paramètres :

- un numéro de session,
- le pointeur sur les données à signer,
- la longueur de ces données,
- un pointeur sur la zone qui va recevoir la signature,
- un pointeur sur la zone qui recevra la longueur de la signature.

Selon le mécanisme de signature choisi, les données doivent avoir une longueur dépendante de la longueur *k* de la clé de signature (Tableau 5 - 11).

Mécanisme	Type de clé	Longueur de la donnée (bytes)	Longueur de la signature (bytes)
PKCS#1 RSA	Privée RSA	<= k-11	k
ISO/IEC 9796 RSA	Privée RSA	<= val. inf. (k/2)	k
X.509 RSA	Privée RSA	<= k	k

Tableau 5 - 11 : Contraintes liées aux mécanismes de signature.

`C_VerifyInit`
Cette fonction initialise l'opération de vérification. La signature étant un appendice de la donnée.

L'opération de vérification n'est « active » que jusqu'au moment où vous appelez `C_Verify`,`C_VerifyUpdate` ou `C_VerifyFinal`. Vous devez alors rappeler la fonction `C_VerifyInit` pour réinitialiser l'opération de vérification.

Vous ne pouvez réaliser les fonctions `C_Verify`, `C_VerifyUpdate`, `C_VerifyFinal` qu'après avoir appelé cette fonction.

Cette fonction ne peut pas initialiser une nouvelle opération de vérification si une autre est toujours active.

Elle reçoit comme paramètres :

- un numéro de session,
- le type de mécanisme de vérification (Tableau 5 - 12),
- le numéro de la clé de vérification qui représente le numéro de l'objet contenant cette clé.

Mécanisme	Type de clé
PKCS#1 RSA (1)	Publique RSA
ISO/IEC 9796 RSA (1)	Publique RSA
X.509 RSA (1)	Publique RSA
DSA (1)	Publique DSA
RC2-MAC	RC2
DES-MAC	DES
triple-DES (mode ECB et CBC)	double ou triple DES

Tableau 5 - 12 : Mécanismes de vérification.

(1) Uniquement pour des vérifications en un seul bloc.

`C_Verify`
Cette fonction vérifie des données en un seul bloc; la signature étant un appendice des données. L'opération de vérification doit avoir été initialisée à l'aide de la fonction `C_SignInit` avant tout appel à la fonction `C_Verify`.

Elle reçoit comme paramètres :

- un numéro de session,
- un pointeur sur la signature à vérifier,
- la longueur (en bytes) de la signature à vérifier,
- un pointeur sur la donnée et la longueur (en bytes) de la donnée.

Selon le mécanisme de vérification choisi, les données doivent avoir une longueur dépendante de la longueur *k* de la clé de vérification (Tableau 5 - 13).

Mécanisme	Type de clé	Longueur de la donnée (bytes)	Longueur de la signature (bytes)
PKCS#1 RSA	Publique RSA	<= k-11	pas applicable
ISO/IEC 9796 RSA	Publique RSA	<= val. inf. (k/2)	pas applicable
X.509 RSA	Publique RSA	<= k	pas applicable
DSA	Publique DSA	20	pas applicable
RC2-MAC	RC2	quelconque	4
DES-MAC	DES	quelconque	4
Triple DES	2-DES ou 3-DES	quelconque	4

Tableau 5 - 13 : Contraintes liées aux mécanismes de vérification.

`C_VerifyUpdate`

Cette fonction continue une opération de vérification en plusieurs parties. L'opération de vérification doit avoir été initialisée à l'aide de la fonction `C_VerifyInit` avant tout appel à la fonction `C_VerifyUpdate`.

Elle reçoit comme paramètres :

- un numéro de session,
- un pointeur sur la donnée,
- la longueur (en bytes) de la donnée.

Selon le mécanisme de vérification choisi, les données doivent avoir une longueur dépendante de la longueur *k* de la clé de vérification (Tableau 5 - 9).

`C_VerifyFinal`

Cette fonction termine une opération de vérification en plusieurs parties, vérifiant ainsi la signature. L'opération de vérification doit avoir été initialisée à l'aide de la fonction `C_VerifyInit` avant tout appel à la fonction `C_VerifyFinal`.

Elle reçoit comme paramètres :

- un numéro de session,
- un pointeur sur la signature,
- la longueur (en bytes) de la signature.

Les contraintes sur la longueur des données sont identiques à celles de la fonction précédente (Tableau 5 - 9).

`C_VerifyRecoverInit`

Cette fonction initialise une opération de vérification; les données peuvent être retrouvées à partir de la signature.

L'opération de vérification n'est « active » que jusqu'au moment où vous appelez `C_VerifyRecover`. Vous devez alors rappeler la fonction `C_VerifyRecoverInit` pour réinitialiser l'opération de vérification.

Vous ne pouvez réaliser les fonctions `C_VerifyRecover` qu'après avoir appelé cette fonction.

Cette fonction ne peut pas initialiser une nouvelle opération de vérification si une autre est toujours active.

Elle reçoit comme paramètres :

- un numéro de session,
- le type de mécanisme de vérification (Tableau 5 - 14),
- le numéro de la clé de vérification qui représente le numéro de l'objet contenant cette clé.

Mécanisme	Type de clé
PKCS#1 RSA	Publique RSA
ISO/IEC 9796 RSA	Publique RSA
X.509 RSA	Publique RSA

Tableau 5 - 14 : Mécanismes de vérification.

`C_VerifyRecover`

Cette fonction vérifie des données en une seule opération; les données peuvent être retrouvées à partir de la signature.

L'opération de vérification doit avoir été initialisée à l'aide de la fonction `C_VerifyRecoverInit` avant tout appel à la fonction `C_VerifyRecover`.

Elle reçoit comme paramètres :

- un numéro de session,
- le pointeur sur la signature,
- la longueur de la signature,
- un pointeur sur la zone qui va recevoir la donnée vérifiée,
- un pointeur sur la zone qui recevra la longueur de la donnée vérifiée.

Selon le mécanisme de vérification choisi, la signature doit avoir une longueur dépendante de la longueur *k* de la clé de vérification (Tableau 5 - 15).

Mécanisme	Type de clé	Longueur de la donnée (bytes)	Longueur de la signature (bytes)
PKCS#1 RSA	Publique RSA	k	<= k-11
ISO/IEC 9796 RSA	Publique RSA	k	<= val. inf. (k/2)
X.509 RSA	Publique RSA	k	<= k

Tableau 5 - 15 : Contraintes liées aux mécanismes de vérification.

5.4.7 Gestion des clés

C_GenerateKey

Cette fonction génère une clé secrète et crée un objet de type clé.

Elle reçoit comme paramètres :

- un numéro de session,
- le type de mécanisme de génération (Tableau 5 - 16),
- un pointeur sur la description de la nouvelle clé,
- le nombre d'attributs de cette description,
- un pointeur qui contiendra le numéro de l'objet accueillant la clé.

Mécanisme	Type de clé
RC2	RC2
RC4	RC4
DES	DES
double-DES	double DES
triple-DES	triple DES

Tableau 5 - 16 : Mécanismes de génération de clés secrètes.

C_GenerateKeyPair

Cette fonction génère une paire de clés publique et secrète et crée des objets de type clé.

Elle reçoit comme paramètres :

- un numéro de session,
- le type de mécanisme de génération (Tableau 5 - 17),
- un pointeur sur la description de la clé publique,
- le nombre d'attributs de cette description,
- un pointeur sur la description de la clé secrète,
- le nombre d'attributs de cette description,
- un pointeur qui contiendra le numéro de l'objet accueillant la clé publique,
- un pointeur qui contiendra le numéro de l'objet accueillant la clé secrète.

Mécanisme	Type de clé
PKCS#1 RSA	Publique et Privée RSA
DSA	DSA publique & privée
PKCS#3 Diffie-Helleman	DH publique & privée

Tableau 5 - 17 : Mécanismes de génération de paires de clés.

5.4.8 Génération de nombres aléatoires

C_SeedRandom

Cette fonction ajoute une *semence* (*seed*) au générateur de nombres aléatoires.

Elle reçoit comme paramètres :

- le numéro de la session,
- un pointeur sur la zone contenant la semence,
- la longueur (en bytes) de cette semence.

C_GenerateRandom
Cette fonction génère un nombre aléatoire.

Elle reçoit comme paramètres :

- le numéro de la session,
- un pointeur sur la zone qui recevra le nombre généré,
- la longueur (en bytes) du nombre à générer.

5.5 Protocole à l'aide de PKCS#11

Au cours ce cette section, nous allons remplacer les différentes fonctions du protocole par les fonctions de la librairie *Cryptoki*. Pour des raisons de clarté, nous ne reprendrons pas dans les fonctions cryptographiques certains paramètres tels que la longueur des données à chiffrer ou encore le numéro de la session. Ces paramètres seront introduits dans les fonctions au cours de la présentation des spécifications (chapitre 6).

5.5.1 Les orientations choisies

Nous avons préféré la fonction **C_SignRecover** à la fonction **C_Sign** car nous ne voulions pas que les données à signer apparaissent dans le résultat de la signature. En effet, la fonction **C_Sign** concatène à la signature, les données à partir desquelles la signature a été effectuée. En utilisant la fonction **C_SignRecover**, les données sont chiffrées et n'apparaissent plus dans la signature.

Le protocole étant basé sur un cryptosystème asymétrique, il nous fallait un mécanisme de chiffrement - déchiffrement et signature - vérification répondant à cette exigence. Nous avons donc opté pour le mécanisme PKCS#1 RSA.

5.5.2 La certification d'une carte

La certification d'une carte est représentée à la Figure 5 - 5.
L'émetteur de la carte dispose d'une paire de clés, Pk_e sa clé publique et Sk_e sa clé secrète.

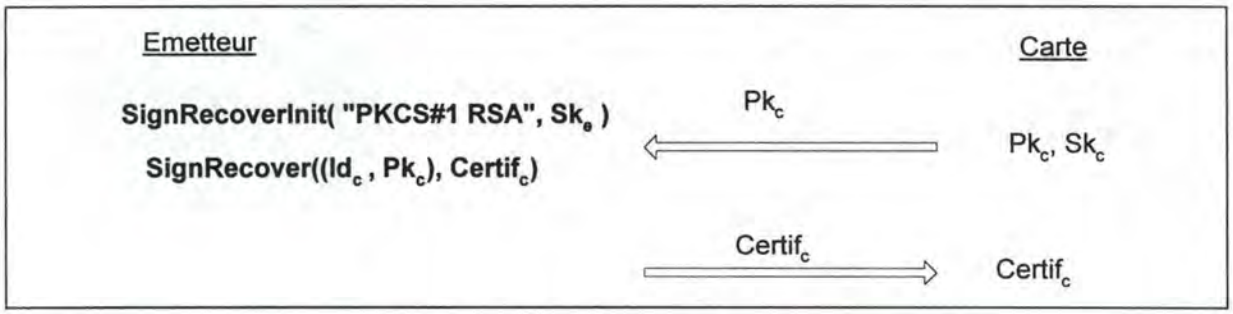


Figure 5 - 5 : Certification d'une carte.

L'émetteur reçoit de la carte sa clé publique Pk_c, il lui attribue un identifiant Id_c et signe le tout avec sa clé secrète Sk_e grâce à la fonction **C_SignRecover**. Il aura au préalable initialisé l'opération de signature à l'aide de la fonction **C_SignRecoverInit**. Le résultat est un certificat C_c qui est envoyé à la carte.

5.5.3 La certification d'un prestataire de services

La certification d'un prestataire de services est représentée à la Figure 5 - 6.
L'émetteur dispose d'une paire de clés, Pk_e sa clé publique et Sk_e sa clé secrète.

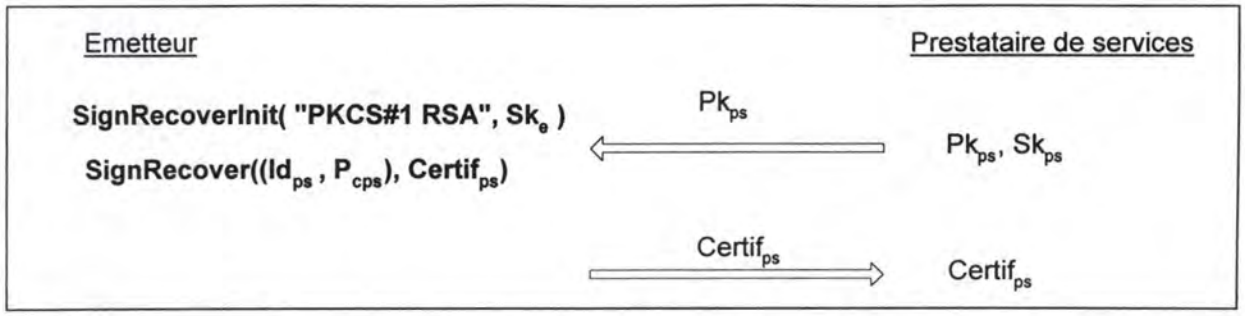


Figure 5 - 6 : Certification d'un prestataire de services.

L'émetteur reçoit du prestataire de services sa clé publique Pk_{ps}, il lui attribue un identifiant Id_{ps} et signe le tout avec sa clé secrète Sk_e grâce à la fonction **C_SignRecover**. Il aura au préalable initialisé l'opération de signature à l'aide de la fonction **C_SignRecoverInit**. Le résultat est un certificat C_{ps} qui est envoyé au prestataire de services.

5.5.4 La certification d'un client

La certification d'un client est représentée à la Figure 5 - 7.

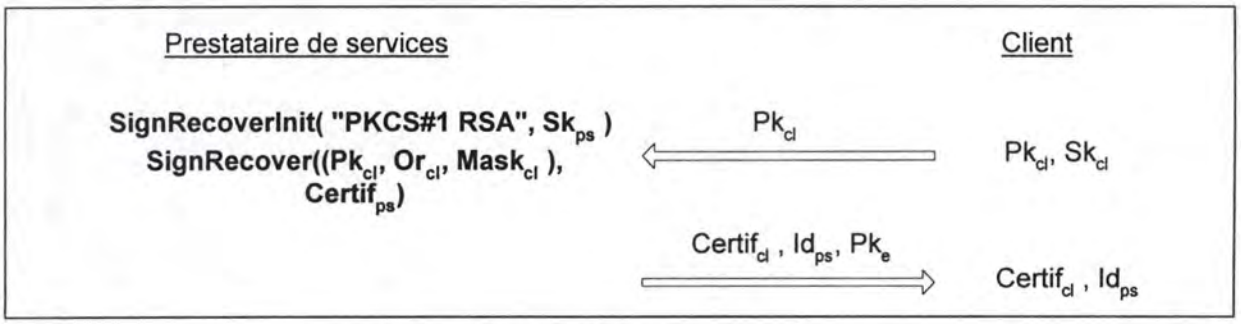


Figure 5 - 7 : Certification d'un client.

Le prestataire de services reçoit du client sa clé publique Pk_{cl}. Il crée le certificat composé de la clé publique Pk_{cl}, de l'Objet Reference Or_{cl}, et du $Mask_{cl}$, le tout étant signé avec sa clé secrète Sk_{ps} grâce à la fonction **C_SignRecover**. Il aura au préalable initialisé l'opération de signature à l'aide de la fonction **C_SignRecoverInit**. Le résultat est alors un certificat C_{cl}. Il est envoyé au client, accompagné de l'identifiant du prestataire de services Id_{ps} ainsi que de la clé publique de l'émetteur qui a certifié Pk_e le prestataire de services.

5.5.5 Authentification carte - prestataire de services

L'authentification entre une carte et un prestataire de services est représentée à la Figure 5 - 8.

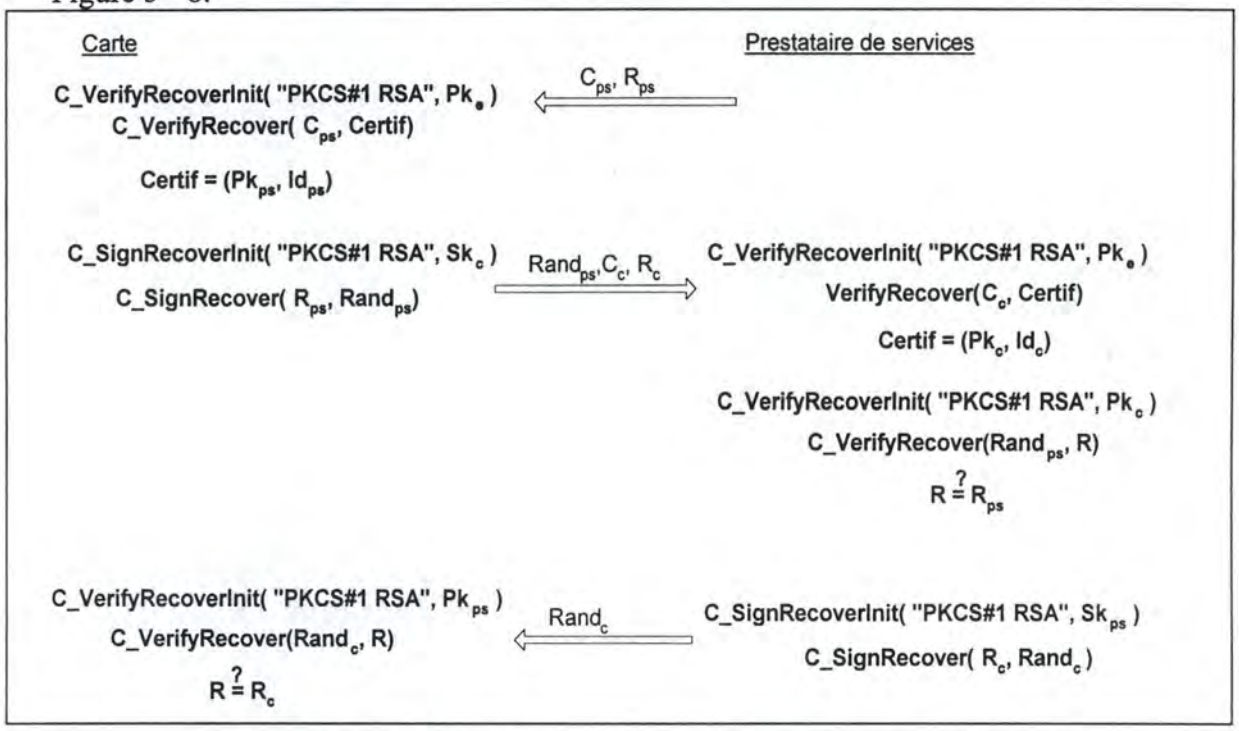


Figure 5 - 8 : Authentification Carte - Prestataire de services.

La carte reçoit du prestataire de services son certificat C_{ps} ainsi qu'un nombre aléatoire R_{ps}. Grâce à la fonction **C_VerifyRecover**, elle effectue alors la vérification de ce certificat à l'aide de la clé publique de son émetteur Pk_e. Elle récupère par conséquent la clé publique Pk_{ps} et l'identifiant Id_{ps} du prestataire de services.

Grâce à la fonction **C_SignRecover**, la carte va maintenant signer le nombre aléatoire R_{ps} à l'aide de sa clé secrète Sk_c. Le résultat est $Rand_{ps}$. Elle envoie alors, au prestataire de services, son certificat C_c, un nombre aléatoire R_c et le nombre aléatoire chiffré $Rand_{ps}$.

Le prestataire de services va alors vérifier le certificat de la carte grâce à la fonction **C_VerifyRecover,** à l'aide de la clé publique de l'émetteur Pk_e. Il récupère ainsi la clé publique Pk_c et l'identifiant Id_c de la carte. La seconde étape consiste à vérifier, grâce à la fonction **C_VerifyRecover**, le nombre aléatoire $Rand_{ps}$ à l'aide de la clé publique de la carte Pk_c. Le résultat de cette vérification sera alors comparé au nombre aléatoire R_{ps} qu'il avait transmis à la carte lors de la première étape.
Le protocole peut continuer si les deux nombres sont égaux. On dit alors que le prestataire de services a authentifié la carte.

Grâce à la fonction **C_SignRecover**, le prestataire de services va ensuite signer, à l'aide de sa clé secrète Sk_{ps}, le nombre aléatoire R_c qu'il vient de recevoir de la carte. Le résultat $Rand_c$ est envoyé à la carte.

Grâce à la fonction **C_VerifyRecover**, la carte va vérifier ce nombre aléatoire $Rand_c$ à l'aide de la clé publique Pk_{ps} du prestataire de services. Le résultat sera comparé avec le nombre aléatoire R_c qu'elle avait transmis au prestataire de services lors de la première étape. La carte aura authentifié le prestataire de services si ces deux nombres sont égaux.

5.5.6 Authentification carte - client

L'authentification entre une carte et un prestataire de services est représentée à la Figure 5 - 9.

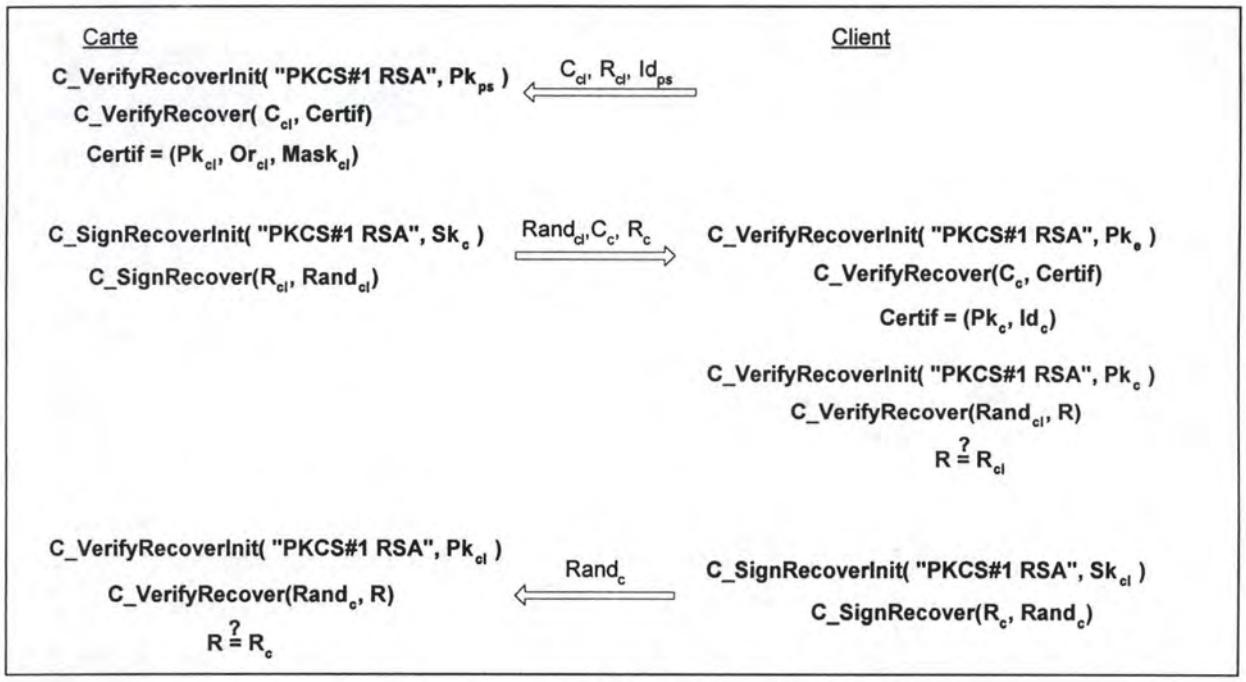


Figure 5 - 9 : Authentification Carte - Client.

Les fonctions cryptographiques utilisées dans ce protocole sont identiques à celles présentées dans la sous-section précédente (authentification carte - prestataire de services). Par conséquent, nous ne les présenterons pas.

Chapitre 6 : Les spécifications

Sommaire

6.1 Le modèle du système

La librairie Cryptoki va être utilisée par les quatre acteurs (carte, prestataire de services, client et émetteur). Nous avons choisi le modèle représenté à la Figure 6 - 1.

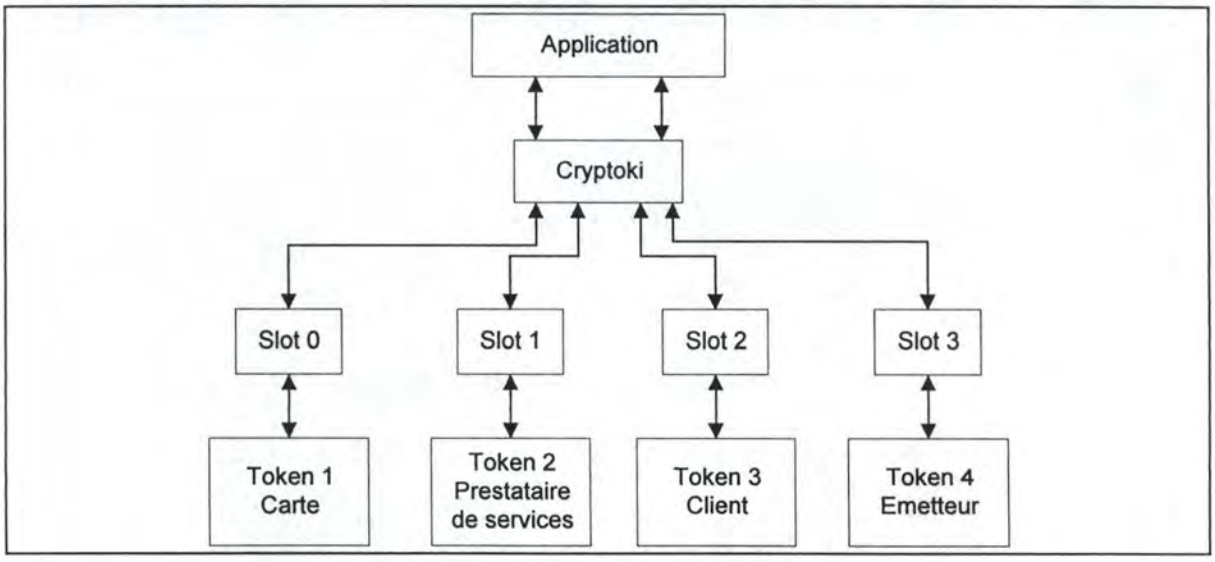
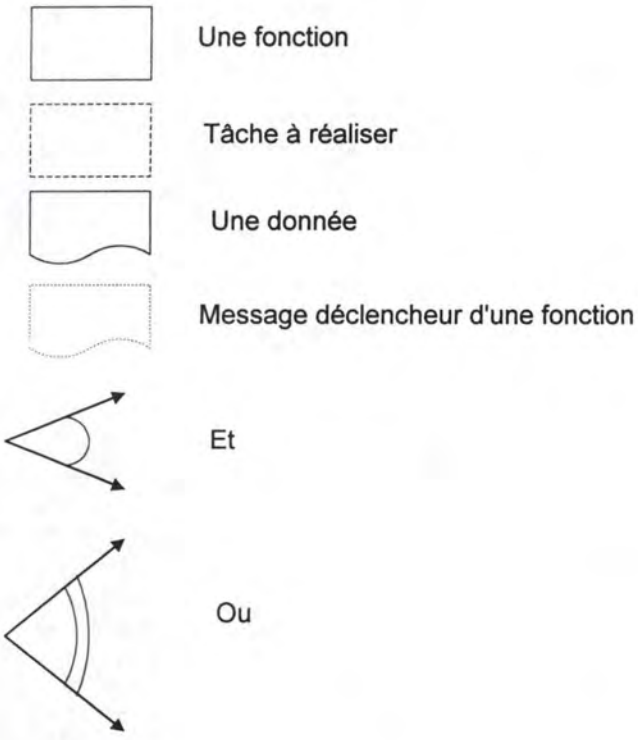


Figure 6 - 1 : Modèle du système.

L'application, pilotant les trois acteurs, sera placée au sommet du modèle. Elle utilise la librairie *Cryptoki* qui réalise le lien, au travers de numéros de session, entre les acteurs et l'application. Chaque acteur est alors identifié par son numéro de session.

Au cours des sections et sous-sections que nous abordons ci-dessous, nous illustrons nos commentaires à l'aide de graphiques. Afin de faciliter la compréhension de ceux-ci, nous fixons les conventions que voici :



6.2 Création des acteurs

La création des acteurs se déroule en trois phases (Figure 6 - 2). Nous initialisons d 'abord l'acteur. Nous créons ensuite sa clé publique et terminons par la création de sa clé privée.

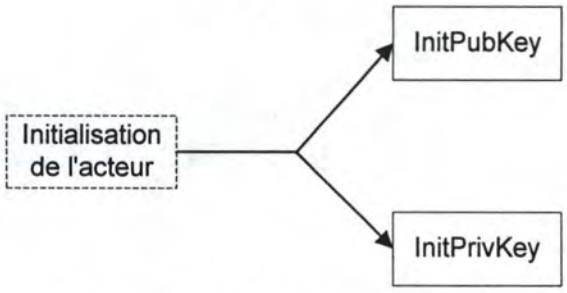


Figure 6 - 2 : Création des acteurs.

6.2.1 L'initialisation de l'acteur

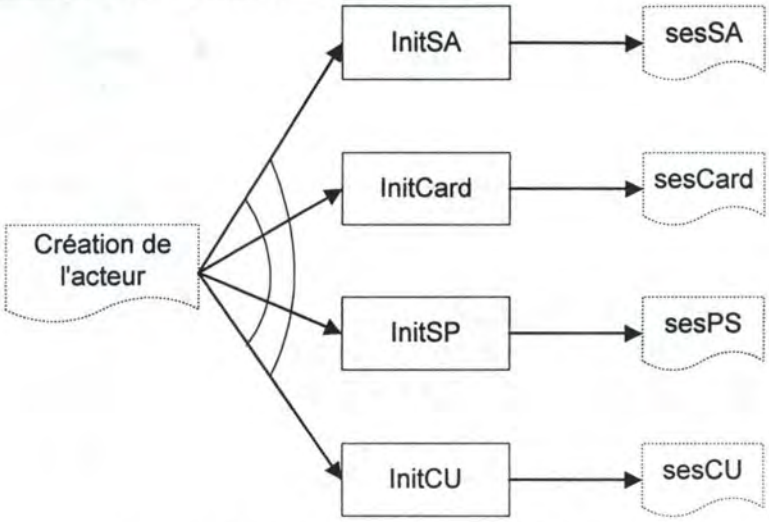


Figure 6 - 3 : Initialisation de l'acteur.

L'initialisation (Figure 6 - 3) de l'acteur est réalisée par les fonctions :

- InitCard pour la carte,
- InitSP pour le prestataire de services,
- InitCU pour le client,
- InitSA pour l'émetteur.

Ces fonctions renvoient un numéro de session :

- sesCard pour la carte,
- sesPS pour le prestataire de services,
- sesCU pour le client,
- sesSA pour l'émetteur.

Le but de ces fonctions est d'attribuer un slot et un token à chaque acteur et un numéro de session permettant d'identifier l'acteur.

Les quatre fonctions sont identiques quant aux fonctions cryptographiques utilisées mais diffèrent du point de vue des données. C'est pourquoi, nous n'indiquerons que le code de la fonction ***InitCard***.

Nous utilisons pour l'attribution d'un slot et d'un token à un acteur la fonction **C_InitToken**. Quant à l'ouverture d'une session, elle sera réalisée grâce à la fonction **C_Opensession**.

Soient :

- slotID = 0, l'identifiant du slot contenant le token,
- PIN = « code secret », le numéro d'identification personnel du « superviseur »,
- sizeof(PIN), la longueur de ce numéro,
- label = « token card », le nom que l'on donne au token,
- Type_de_session = « Read / Write session », le type de session que l'on souhaite ouvrir,

- sesCard, le numéro de session de la carte (numéro attribué par la fonction C_Opensession),
- res, l'état final de l'exécution des fonctions **C_Opensession** et **C_InitToken**.

Le corps peut alors s'écrire :

```
slotID = 0
PIN = « code secret »
label = « token card »
Type_de_session = « Read / Write session »
res = C_InitToken(slotID, PIN, sizeof(PIN), label)
res = C_Opensession(slotID,Type_de_session,sesCard)
```

6.2.2 La création des clés de l'acteur

Nous avons choisi des clés d'une longueur de 512 bits tout en sachant que l'application peut supporter des clés d'une longueur de 768 ou 1024 bits.

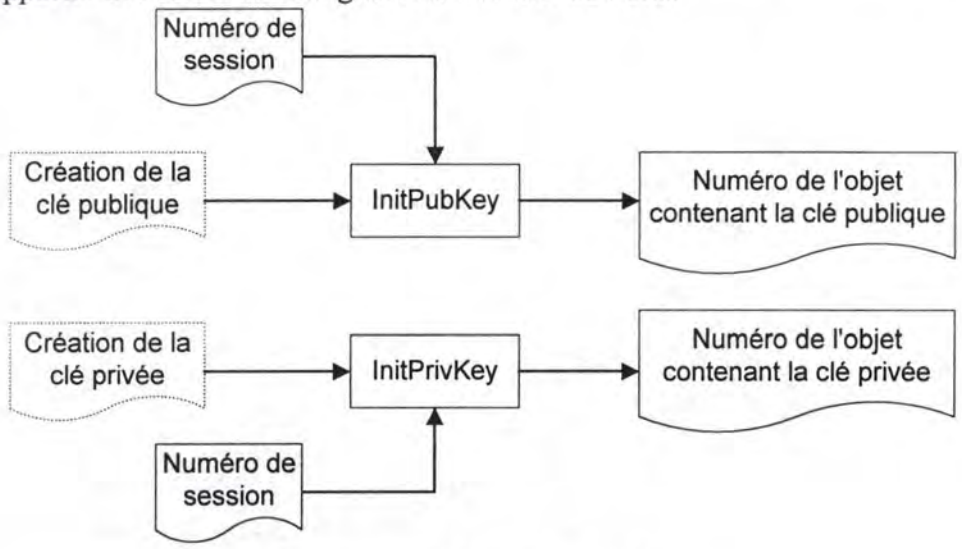


Figure 6 - 4 : Création des clés.

La création des clés (Figure 6 - 4) est réalisée à l'aide de deux fonctions, ***InitPubKey*** s'occupant de la création des clés publiques et ***InitPrivKey*** s'occupant de la création des clés privées.
Ces fonctions ne pourront être appelées, pour un acteur, qu'une seule fois, généralement à la création de l'acteur. Elles sont paramètrées de manière à pouvoir créer une clé quel que soit l'acteur qui en sera propriétaire. C'est pourquoi ces fonctions reçoivent comme paramètre un numéro de session (permettant d'identifier l'acteur), et les attributs et valeurs (reprises dans un descriptif) nécessaires à la création de la clé. Elles renvoient le numéro de l'objet (numéro unique) contenant la clé. Pour ces créations de clés, nous avons employé la fonction **C_CreateObject**.

Soient :

- num_session = sesCU ∨ sesPS ∨ sesCard ∨ sesSA, le numéro de session de l'acteur pour lequel on désire créer les clés;
- num_Pub_Key, le numéro de l'objet qui va contenir la clé publique de l'acteur;

- descriptif_Pub_Key, le descriptif de la clé publique que l'on désire créer :
  - Type_Objet = Public_Key_Object, le type d'objet,
  - Type_Clé = Public_Key_RSA, le type de clé,
  - Label_Clé = « Clé publique de l'acteur X », le label de la clé,
  - ID_Clé = 0000, l'identifiant de la clé,
  - Modulus, le Modulo *n*,
  - Long_Key = 512, la longueur (en bits) de la clé que l'on veut créer,
  - Pub_exponent, l'exposant public *e*;
- nbre_attr_Pub_Key, le nombre d'attributs contenus dans le descriptif de la clé;
- res, l'état final de l'exécution de la fonction **C_CreateObject**.

Le modulo et l'exposant publique sont fournis par une application appelée Calc32 (développée par l'équipe software de Gemplus) dont le but est de calculer les données (stockées dans un fichier) nécessaires la création de tout type de clés.

Le corps de la fonction ***InitPubKey*** peut alors s'écrire :

```
num_session = sesCU ∨ sesPS ∨ sesCard ∨ sesSA
descriptif_Pub_Key = {
                        Type_Objet = Public_Key_Object
                        Type_Clé = Public_Key_RSA
                        Label_Clé = « Clé publique de l'acteur X »
                        ID_Clé = 0000
                        Modulus = {...}
                        Long_Key = 512
                        Pub_exponent = {...}
                    }
nbre_attr_Pub_Key = 7
res = C_CreateObject(num_session,descriptif_Pub_Key, nbre_attr_Pub_Key, num_Pub_Key)
```

Pour la création d'une clé privée, les paramètres présentés ci-dessus sont identiques excepté au niveau du descriptif de la clé.

Soient

- num_Priv_Key, le numéro de l'objet qui va contenir la clé privée de l'acteur;
- descriptif_Priv_Key, le descriptif de la clé privée que l'on désire créer :
  - Type_Objet = Private_Key_Object, le type d'objet,
  - Type_Clé = Private_Key_RSA, le type de clé,
  - Label_Clé = « Clé privée de l'acteur X », le label de la clé,
  - ID_Clé = 0001, l'identifiant de la clé,
  - Modulus, le Modulo *n*,
  - Pub_exponent, l'exposant public *e*.
  - Priv_exponent, l'exposant privé *d*,
  - Prime1, Prime2, les deux nombres premiers *p* et *q*,
  - Exponent1, l'exposant privé *d* modulo *p-1*,
  - Exponent2, l'exposant privé *d* modulo *q-1*,
  - Coeficient, le coefficient q^{-1} modulo *p*;
- nbre_attr_Priv_Key, le nombre d'attributs contenus dans le descriptif de la clé privée;

- res, l'état final de l'exécution des fonctions **C_CreateObject**.

Le modulo *n*, l'exposant public *e*, l'exposant privé *d*, les deux nombres premiers *p* et *q*, l'exposant privé *d* modulo *p-1*, l'exposant privé *d* modulo *q-1*, le coefficient q^{-1} modulo *p* sont fournis par l'application Calc32.

Le corps de la fonction ***InitPrivKey*** peut alors s'écrire :

```
num_session = sesCU ∨ sesPS ∨ sesCard ∨ sesSA
descriptif_Priv_Key = {
                        Type_Objet = Private_Key_Object
                        Type_Clé = Private_Key_RSA
                        Label_Clé = « Clé privée de l'acteur X »
                        ID_Clé = 0001
                        Modulus = {...}
                        Pub_exponent = {...}
                        Priv_exponent = {...}
                        Prime1 = {...}
                        Prime2 = {...}
                        Exponent1 = {...}
                        Exponent2 = {...}
                        Coeficient = {...}
                      }
nbre_attr_Priv_Key = 12
res = C_CreateObject(num_session,descriptif_Priv_Key, nbre_attr_Priv_Key, num_Priv_Key)
```

6.3 Les certifications

Une fois qu'un acteur a été créé, il lui reste à demander un certificat à l'émetteur dans le cas de la carte et au prestataire de services dans le cas du client. Les certificats seront stockés dans des objets *données*.

L'acteur chargé de fournir un certificat devra s'assurer qu'il n'en a pas déjà donné un au demandeur.

La certification se déroule en deux phases (Figure 6 - 5). Le certificat est d'abord créé par l'autorité compétente (émetteur ou prestataire de services); il est ensuite envoyé, avec la clé publique de l'autorité, au demandeur du certificat.

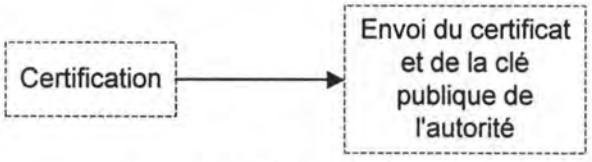


Figure 6 - 5 : Certifications.

La création du certificat, ainsi que l'envoi du certificat, d'une carte et d'un prestataire de services sont différents de ceux d'un client. C'est pourquoi nous avons spécifié deux fonctions de création du certificat et d'envoi que nous détaillons au cours des deux sous-sections ci-dessous.

6.3.1 La carte et le prestataire de services

La carte et le prestataire de services s'adressent tous deux à l'émetteur de la carte. Le certificat est identique pour les deux acteurs du point de vue du type des attributs.

La création d'un certificat (Figure 6 - 6) est réalisée par la fonction ***Certif***, dont l'objectif est de simuler l'émetteur d'une carte, le travail de cet émetteur étant de signer les données qu'il reçoit. Une fois ce certificat créé, il est transmis à son propriétaire grâce à la fonction ***EnvoiCard*** (lorsque le propriétaire est une carte) et ***EnvoiSP*** (lorsque le propriétaire est un prestataire de services).

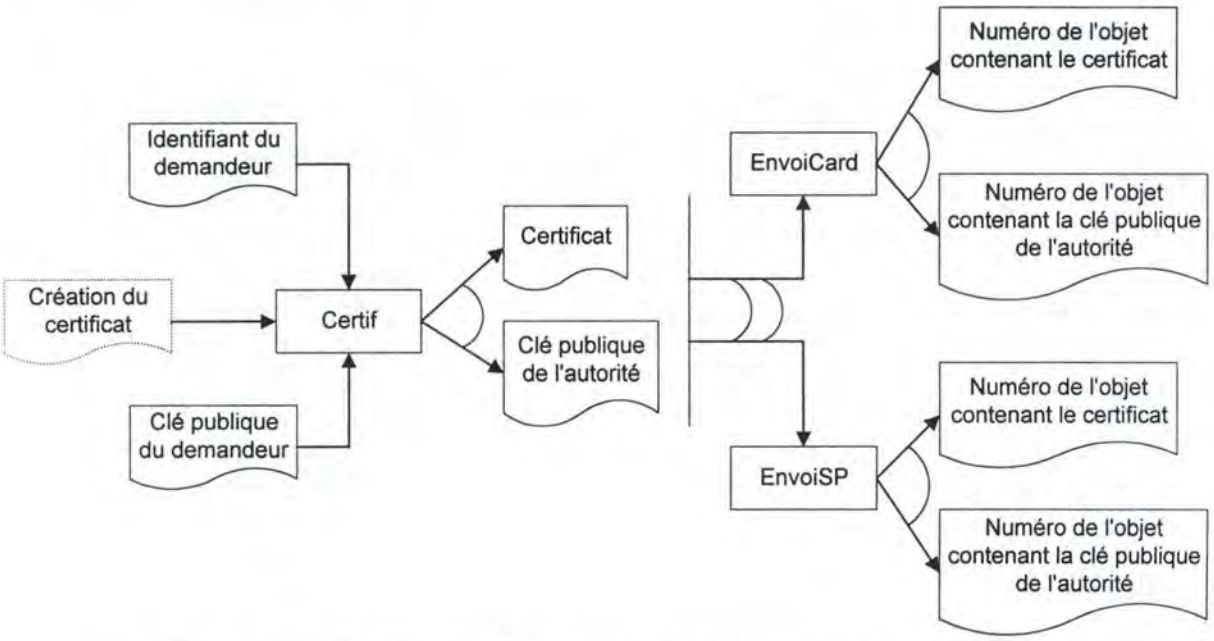


Figure 6 - 6 : Certification d'une carte et d'un prestataire de services.

La fonction ***Certif*** reçoit la clé publique du demandeur. Elle va créer un certificat en signant, avec la clé secrète de l'émetteur, la donnée reçue (clé publique du demandeur) concaténée avec un identifiant. Vu les contraintes liées à la taille des données qu'il est possible de signer, nous avons été obligés de scinder le couple (clé publique du demandeur, identifiant) en blocs (selon les contraintes). Les blocs sont alors signés et reconcaténés les uns aux autres afin de donner comme résultat le certificat final. La signature est effectuée à l'aide de la fonction **C_SignRecover**.

La fonction ***EnvoiCard*** ou ***EnvoiSP*** matérialise l'échange d'un certificat et d'une clé publique entre un émetteur et un demandeur. Cette fonction aura pour but de créer un objet *données* qui contiendra le certificat qu'elle reçoit et un objet *clé publique* contenant le clé publique reçue de l'émetteur. Pour effectuer ces deux opérations, nous avons employé la fonction **C_CreateObject**.

Il nous faut apporter une précision en ce qui concerne la fonction ***EnvoiCard*** et ***EnvoiSP***. Notre application accepte d'utiliser des cartes et des prestataires de services « pirates ». Une carte ou prestataire de services « pirate » dispose d'un certificat qu'elle/il a fabriqué elle/lui-même (certificat bidon). Le but de cette manoeuvre étant de prouver que le protocole s'arrête lorsqu'un acteur « pirate » se présente. L'application accepte également

qu'un acteur « pirate » se convertisse en se faisant connaître auprès d'un émetteur par le biais d'une demande de certificat.
Vu qu'un acteur ne peut pas disposer à un moment donné de deux certificats, les fonctions ***EnvoiCard*** et ***EnvoiSP*** seront chargées de vérifier, au moyen de la fonction **C_FindObjects**, si le demandeur ne possède pas un certificat. Un éventuel faux certificat sera détruit au moyen de la fonction **C_DestroyObject**.

En résumé, la fonction ***Certif*** vérifiera qu'un acteur n'a pas déjà demandé un certificat et les fonctions ***EnvoiCard*** et ***EnvoiSP*** détruiront le faux certificat d'un acteur « pirate » ayant demandé un certificat à un émetteur.

Décomposons nos spécifications en deux parties : la première est consacrée à la fonction ***Certif*** et la seconde à la fonction ***EnvoiCard***. La fonction ***EnvoiSP*** étant identique à la fonction ***EnvoiCard***.

6.3.1.1 La fonction Certif

Soient :

- Pub_Key, la clé publique du demandeur,
- ID, l'identifiant du demandeur attribué par l'émetteur,
- Session = sesSA, le numéro de session de l'émetteur,
- Type_Mec = PCKS#1_RSA, le type de mécanisme de signature,
- Clé_Sign, le numéro de la clé de signature qui représente le numéro de l'objet contenant cette clé,
- Donnée = (Pub_Key + ID), la donnée à signer,
- Res_Sign, le résultat de la signature,
- res, l'état final de l'exécution de la fonction **C_SignRecoverInit** et **C_SignRecover**.

Comme nous l'avons dit précédemment, nous devrons couper en blocs la Donnée.
Soient :

- $Bloc_i$, le $i^{ème}$ bloc de la Donnée,
- Sizeof($Bloc_i$), la longueur du $i^{ème}$ bloc de la Donnée,
- R_Sign_i, le résultat de la signature du $i^{ème}$ bloc,
- $Long\_R\_Sign_i$, la longueur de la signature du $i^{ème}$ bloc,
- i = 1,...,n.

Le corps de la fonction ***Certif*** peut alors s'écrire :

```
session = sesSA
Type_Mec = PCKS#1_RSA
Donnée = (Pub_Key + ID)
Tant que i <= n
   Bloc_i = {...}
   res = C_SignRecoverInit(session, Type_Mec, Clé_Sign)
   res = C_SignRecover(session, Bloc_i, sizeof(Bloc_i), R_Sign_i , Long_R_Sign_i)
   Res_Sign = Res_Sign + R_Sign_i
   i = i +1
```

6.3.1.2 La fonction EnvoiCard

Soient :

- session = sescard, le numéro de la session;
- num_Certif, le numéro de l'objet qui va contenir le certificat de la carte;
- num_Pub_Key, le numéro de l'objet qui va contenir la clé publique de l'émetteur;
- descriptif_Pub_Key_SA, le descriptif de l'objet qui va contenir la clé publique de l'émetteur :
  - Type_Objet = Public_Key_Object, le type d'objet,
  - Type_Clé = Public_Key_RSA, le type de clé,
  - Label_Clé = « Clé publique de l'émetteur », le label de la clé,
  - ID_Clé = 0010, l'identifiant de la clé,
  - Modulus, le Modulo *n* transmis par l'émetteur,
  - Long_Key = 512, la longueur (en bits) de la clé que l'on veut créer,
  - Pub_exponent, l'exposant public *e* transmis par l'émetteur;
- nbre_attr_descr_Pub_Key_SA, le nombre d'attributs contenus dans descriptif_Pub_Key_SA;
- descriptif_Objet_rech, le descriptif de l'objet *données* que l'on désire retrouver :
  - Type_Objet = Data_Object, le type de l'objet,
  - Label_Objet = « certificat bidon », le label de l'objet recherché;
- nbre_attr_descriptif_Objet_rech, le nombre d'attributs de ce descriptif;
- descriptif_certificat, le descriptif de l'objet qui va accueillir le certificat :
  - Type_Objet = Data_Object, le type de l'objet,
  - Label_Objet = « certificat de la carte », le label de l'objet que l'on crée,
  - carac_objet = public, le caractère de l'objet,
  - Res_Sign, le certificat obtenu après exécution de la fonction ***Certif***,
  - appli = « application », le label de l'application qui gère l'objet contenant le certificat;
- nbre_attr_descriptif_certificat,le nombre d'attributs de ce descriptif;
- Liste_Objet, la liste des numéros des objets répondant au descriptif;
- nbre_max_Objet, le nombre maximum d'objets que l'on souhaite recevoir,
- num_dern_Objet, le numéro du dernier objet trouvé;
- res, l'état final de l'exécution des fonctions **C_FindObjectInit**, **C_FindObjects** et **C_CreateObject**.

Il nous faut apporter des précisions quant à certains descriptifs. Une fois qu'une clé est créée, il n'y a pas moyen d'en connaître sa valeur. Or, dans le protocole, des clés sont échangées. Nous avons donc adopté comme solution l'échange des données nécessaires à la recréation de la clé. Le récepteur de ces données, afin de pouvoir disposer de la clé, n'a plus qu'à exécuter la fonction **C_CreateObject** après avoir introduit ces données dans un descriptif. Les certificats seront de la forme (identifiant, modulo, exposant public), le modulo et l'exposant permettant de reconstruire la clé publique.

La fonction recevra un certificat ainsi qu'un modulo et un exposant public lui permettant de reconstruire la clé publique de l'émetteur.

Le corps de la fonction ***EnvoiCard*** peut alors s'écrire :

```
session = sescard
Recherche d'un éventuel certificat bidon
descriptif_Objet_rech = {
                            Type_Objet = Data_Object
                            Label_Objet = « certificat bidon »
                        }
nbre_attr_descriptif_Objet_rech = 2
nbre_max_Objet = 1
res = C_FindObjectInit(session, descriptif_Objet_rech, nbre_attr_descriptif_Objet_rech)
res = C_FindObjects(session, Liste_Objet, nbre_max_Objet, num_dern_Objet)

Destruction de l'objet s'il existe
Si num_dern_Objet = 1 Alors res = C_DestroyObject(session, Liste_Objet)
Création de l'objet qui va contenir le certificat
descriptif_certificat = {
                            Type_Objet = Data_Object,
                            Label_Objet = « certificat de la carte »,
                            carac_objet = public,
                            Res_Sign,
                            appli = « application »
                        }
nbre_attr_descriptif_certificat = 5
res = C_CreateObject(session, descriptif_certificat, nbre_attr_descriptif_certificat, num_Certif)

Création de l'objet qui va contenir la clé publique de l'émetteur
descriptif_Pub_Key_SA = {
                            Type_Objet = Public_Key_Object
                            Type_Clé = Public_Key_RSA
                            Label_Clé = « Clé publique de l'émetteur »
                            ID_Clé = 0010
                            Modulus = {...}
                            Long_Key = 512
                            Pub_exponent = {...}
                        }
nbre_attr_descr_Pub_Key_SA = 7
res = C_CreateObject(session, descriptif_Pub_Key_SA, nbre_attr_descr_Pub_Key_SA,
num_Pub_Key)
```

6.3.2 Le client

Deux fonctions sont utilisées, ***CertifCU***, dont le but est de créer le certificat du client, et ***EnvoiCU***, dont le but est de créer l'objet contenant le certificat venant du prestataire de services ainsi que deux objets, l'un contenant la clé publique du prestataire et l'autre la clé publique de l'émetteur qui a certifié le prestataire.

Les spécifications de ces fonctions sont identiques aux précédentes à ceci près :

- Le certificat créé par la fonction ***CertifCU*** comprend la clé publique, le Mask et l'Objet Reference du client.

- La fonction ***EnvoiCU*** crée, en plus du certificat et de la clé publique du prestataire de services, un objet *données* qui contiendra l'identifiant du prestataire de services qui a émis le certificat. Elle détruit également un éventuel certificat et identifiant bidon. Généralement l'identifiant bidon sera l'identifiant d'un prestataire de services existant.

Une table de correspondance reprenant le couple (nom du prestataire, numéro de l'objet contenant le certificat émis par ce prestataire) est mise à jour à chaque nouvelle création d'un certificat. Il en est de même pour la table reprenant le couple (nom du prestataire, numéro de l'objet contenant l'identifiant du prestataire qui a émis le certificat).

6.4 Les authentifications

Comme nous l'avons vu dans le chapitre 3, le protocole définit deux types d'authentification (Figure 6 - 7) : d'une part, l'authentification entre une carte et un prestataire de services et d'autre part, l'authentification entre une carte et un client que nous détaillons au cours des sous-sections ci-dessous. Les authentifications sont indépendantes, l'une pouvant être réalisée sans que l'autre n'ai été effectuée au préalable.

Figure 6 - 7 : Les authentifications.

6.4.1 L'authentification Carte - Prestataire de services

Afin d'expliquer clairement ce que réalise chaque fonction, reprenons la représentation du protocole d'authentification Carte - Prestataire de services (Figure 6 - 8).

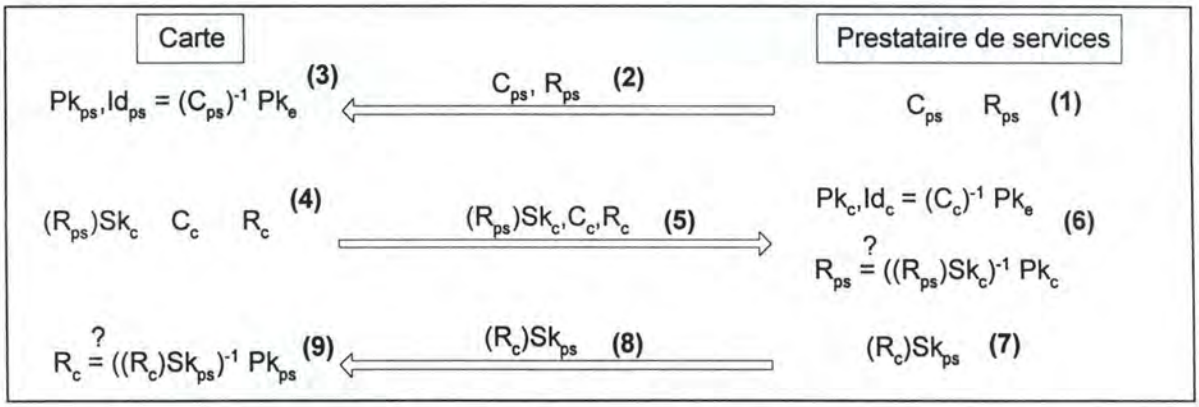


Figure 6 - 8 : Protocole d'authentification Carte - Prestataire de Services.

Cette sous-section a pour but de spécifier la fonction ***AuthentifCSP*** (Figure 6 - 9) qui matérialise l'authentification mutuelle d'une carte et d'un prestataire de services. Cette fonction va réaliser le protocole de la Figure 6 - 8.

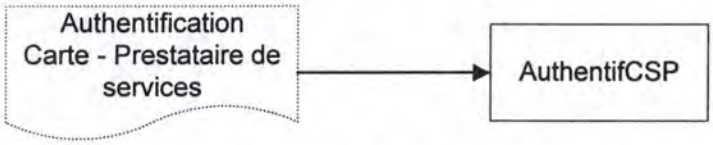


Figure 6 - 9 : Authentification Carte-Prestataire de services.

Cette fonction est découpée en quatre parties (Figure 6 - 10) que nous spécifions ci-dessous.

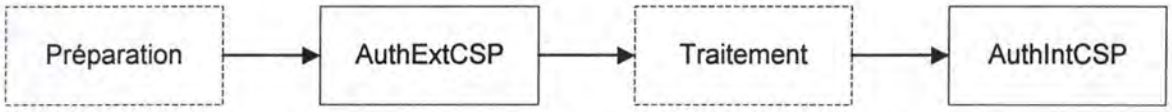


Figure 6 - 10 : Décomposition de la fonction AuthentifCSP.

6.4.1.1 La préparation

La tâche reprise sous le nom de *Préparation* (appartenant à la fonction ***AuthentifCSP***), matérialise le travail du prestataire de services. Elle a pour but de générer un nombre aléatoire à l'aide de la fonction **C_GenerateRandom** et de rechercher parmi tous les objets du prestataire de services celui qui contient son certificat à l'aide des fonctions **C_FindObjects** et **C_GetAttributeValue**. Elle réalise en réalité le point (1) du protocole (Figure 6 - 8). La Figure 6 - 11 représente les tâches réalisées par la *Préparation.*.

Figure 6 - 11 : La Préparation.

Rps est le nombre aléatoire du prestataire de services, *Cps* étant son certificat (Figure 6 - 11).

Soient :

- session = sesPS, le numéro de la session du prestataire de services;
- Rps, le nombre aléatoire généré par le prestataire de services;
- descriptif_certificat, le descriptif de l'objet qui contient le certificat :
  - Type_Objet = Data_Object, le type de l'objet,
  - Label_Objet = « certificat du prestataire de services », le label de l'objet,
  - carac_objet = public, le caractère de l'objet,

  - appli = « application », le label de l'application qui gère l'objet contenant le certificat;
- nbre_attr_descriptif_certificat, le nombre d'attributs de ce descriptif;
- Certif_PS, le numéro de l'objet contenant le certificat;
- res, l'état final de l'exécution des fonctions **C_FindObjectInit**, **C_FindObjects**, **C_GetAttributeValue** et **C_GenerateRandom**;
- nbre_max_Objet, le nombre maximum de numéros d'objet que l'on souhaite recevoir;
- num_dern_Objet, le numéro du dernier objet trouvé;
- descriptif_Certif_Info, le descriptif des attributs (du certificat) pour lesquels on souhaite une valeur :
  - Cps, le certificat trouvé;
- nbre_attr_descriptif_Certif_Info, le nombre d'attributs contenus dans descriptif_Certif_Info.

Le corps de la partie de code de la fonction ***AuthentifCSP*** reprise sous le nom *Préparation* peut alors s'écrire :

```
session = sesPS
Génération du nombre aléatoire du prestataire de services
res = C_GenerateRandom(session, Rps, sizeof(Rps))

Recherche de l'objet contenant le certificat du prestataire de services
descriptif_certificat = {
                            Type_Objet = Data_Object,
                            Label_Objet = « certificat du prestataire de services »,
                            carac_objet = public,
                            appli
                        }
nbre_attr_descriptif_certificat = 4
nbre_max_Objet = 1
res = C_FindObjectsInit(session, descriptif_certificat, nbre_attr_descriptif_certificat)
res = C_FindObjects(session, Certif_PS, nbre_max_Objet, num_dern_Objet)

Extraction de la valeur du certificat
descriptif_Certif_Info = {
                            Cps
                        }
nbre_attr_descriptif_Certif_Info = 1
res = C_GetAttributeValue(session, Certif_PS, descriptif_Certif_Info,
nbre_attr_descriptif_Certif_Info)
```

6.4.1.2 AuthExtCSP

La fonction AuthExtCSP (dont le corps est représenté à la Figure 6 - 12) matérialisant le travail de la carte, a pour but de :

- vérifier la signature du certificat venant du prestataire de services à l'aide de la fonction **C_VerifyRecover** (point (3) de la Figure 6 - 8),
- rechercher, parmi tous ses objets, celui qui contient son certificat à l'aide des fonctions **C_FindObjects** et **C_GetAttributeValue** (point (4) de la Figure 6 - 8),

- générer son nombre aléatoire à l'aide de la fonction **C_GenerateRandom** (point (4) de la Figure 6 - 8),
- signer le nombre aléatoire venant du prestataire de services à l'aide de la fonction **C_SignRecover** (point (4) de la Figure 6 - 8),
- créer un objet contenant la clé publique et l'identifiant du prestataire de services.

Cette fonction reçoit le certificat *Cps* et le nombre aléatoire *Rps* du prestataire de services et renvoie le certificat *Cc*, le nombre aléatoire de la carte *Rc* et le nombre aléatoire du prestataire de services signé. Ce qui est représenté par les points (2) et (5) de la Figure 6 - 8.

Le certificat *Cps* sera décomposé en blocs afin de respecter les contraintes de taille de données lors d'une vérification de signature.

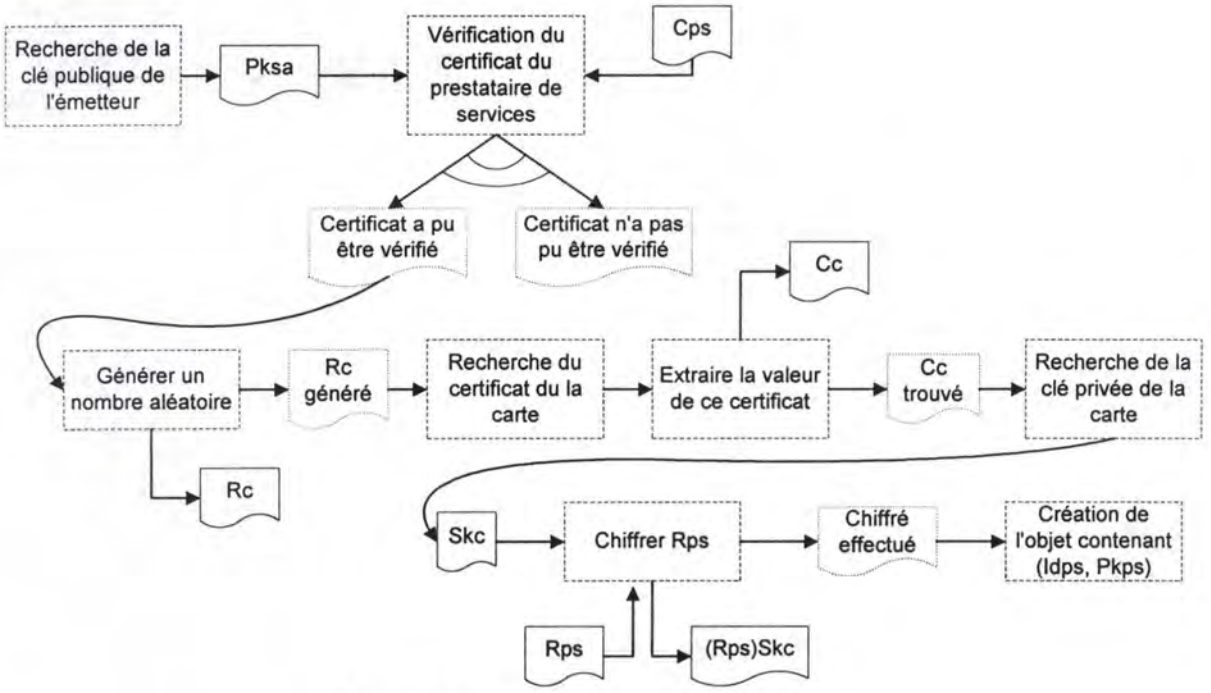


Figure 6 - 12 : Fonction AuthExtCSP.

Soient :

- Pksa, le numéro de l'objet contenant la clé publique de l'émetteur;
- Skc, le numéro de l'objet contenant la clé privée de la carte;
- Rc, le nombre aléatoire de la carte;
- Rps, le nombre aléatoire du prestataire de services;
- Cps, le certificat du prestataire de services;
- Certif_C, le numéro de l'objet contenant le certificat de la carte;
- session = sesCard, le numéro de session de la carte;
- descriptif_Pub_Key_SA, le descriptif de l'objet qui contient la clé publique de l'émetteur :
  - Type_Objet = Public_Key_Object, le type d'objet,
  - Type_Clé = Public_Key_RSA, le type de clé,
  - Label_Clé = « Clé publique de l'émetteur », le label de la clé,
  - ID_Clé = 0010, l'identifiant de la clé,

- nbre_attr_descr_Pub_Key_SA, le nombre d'attributs contenus dans descriptif_Pub_Key_SA;
- nbre_max_Objet, le nombre maximum de numéros d'objets que l'on souhaite recevoir;
- num_dern_Objet, le numéro du dernier objet trouvé;
- Res_Verif, le résultat de la vérification du certificat du prestataire de services;
- res, l'état final de l'exécution des fonctions **C_FindObjectInit**, **C_FindObjects**, **C_GetAttributeValue**, **C_GenerateRandom**, **C_VerifyRecoverInit**, **C_VerifyRecover**, **C_SignRecoverInit** et **C_SignRecover**;
- Type_Mec = PCKS#1_RSA, le type de mécanisme de vérification;
- $Bloc_i$, le $i^{ème}$ bloc du certificat du prestataire de services (Cps);
- Sizeof($Bloc_i$), la longueur du $i^{ème}$ bloc du certificat du prestataire de services;
- R_Verif_i, le résultat de la vérification du $i^{ème}$ bloc;
- $Long\_R\_Verfi_i$, la longueur du résultat de la vérification du $i^{ème}$ bloc;
- i = 1,...,n
- descriptif_certificat, le descriptif de l'objet qui contient le certificat de la carte :
  - Type_Objet = Data_Object, le type de l'objet,
  - Label_Objet = « certificat de la carte », le label de l'objet,
  - carac_objet = public, le caractère de l'objet,
  - appli = « application », le label de l'application qui gère l'objet contenant le certificat;
- nbre_attr_descriptif_certificat, le nombre d'attributs de ce descriptif;
- descriptif_Certif_Info, le descriptif des attributs (du certificat) pour lesquels on souhaite une valeur :
  - Cc, le certificat trouvé;
- nbre_attr_descriptif_Certif_Info, le nombre d'attributs contenus dans descriptif_Certif_Info;
- descriptif_Priv_Key_C, le descriptif de la clé privée (de la carte) que l'on recherche:
  - Type_Objet = Private_Key_Object, le type d'objet,
  - Type_Clé = Private_Key_RSA, le type de clé,
  - Label_Clé = « Clé privée de la carte », le label de la clé,
  - ID_Clé = 0001, l'identifiant de la clé;
- nbre_attr_Priv_Key_C, le nombre d'attributs contenus dans le descriptif de la clé privée;
- R_Rps, le résultat de la signature du nombre aléatoire du prestataire de services;
- Long_R_Rps, la longueur du résultat R_Rps;
- descriptif_Pub_Key_PS, le descriptif de la clé publique que l'on désire créer :
  - Type_Objet = Public_Key_Object, le type d'objet,
  - Type_Clé = Public_Key_RSA, le type de clé,
  - Label_Clé = « Clé publique du prestataire de services», le label de la clé,
  - ID_Clé = Idps, l'identifiant de la clé,
  - Modulus, le Modulo *n*,
  - Long_Key = 512, la longueur (en bits) de la clé que l'on veut créer,
  - Pub_exponent, l'exposant public *e*;
- nbre_attr_Pub_Key_PS, le nombre d'attributs contenus dans le descriptif de la clé;

- num_Pub_Key, le numéro de l'objet qui va contenir (dans la carte) la clé publique du prestataire de services.

Le corps de la fonction peut alors s'écrire :

```
session = sesCard
Recherche de la clé publique de l'émetteur
descriptif_Pub_Key_SA = {
                                Type_Objet = Public_Key_Object
                                Type_Clé = Public_Key_RSA
                                Label_Clé = « Clé publique de l'émetteur »
                                ID_Clé = 0010
                            }
nbre_attr_descr_Pub_Key_SA = 4
nbre_max_Objet = 1
res = C_FindObjectInit(session, descriptif_Pub_Key_SA, nbre_attr_descr_Pub_Key_SA)
res = C_FindObjects(session, Pksa, nbre_max_Objet, num_dern_Objet)

Vérification du certificat du prestataire de services
Type_Mec = PCKS#1_RSA
Tant que i <= n
   Bloc_i = {...}
   res = C_VerifyRecoverInit(session, Type_Mec, Pksa)
   res = C_VerifyRecover(session, Bloc_i, sizeof(Bloc_i), R_Verif_i , Long_R_Verif_i)
   Res_Verif = Res_Verif + R_Verif_i
   i = i +1
Si res <>0
   Alors
      Le certificat du prestataire de services a pu être vérifié

      Générer le nombre aléatoire de la carte
      res = C_GenerateRandom(session, Rc, sizeof(Rc))

      Recherche de l'objet contenant le certificat de la carte
      descriptif_certificat = {
                                Type_Objet = Data_Object,
                                Label_Objet = « certificat de la carte »,
                                carac_objet = public,
                                appli
                            }
      nbre_attr_descriptif_certificat = 4
      nbre_max_Objet = 1
      res = C_FindObjectsInit(session, descriptif_certificat,
      nbre_attr_descriptif_certificat)
      res = C_FindObjects(session, Certif_C, nbre_max_Objet, num_dern_Objet)
      Extraction de la valeur du certificat
      descriptif_Certif_Info = {
                                  Cc
                               }
      nbre_attr_descriptif_Certif_Info = 1
      res = C_GetAttributeValue(session, Certif_C, descriptif_Certif_Info,
      nbre_attr_descriptif_Certif_Info)
```

Rechercher la clé privée de la carte

```
descriptif_Priv_Key_C = {
                        Type_Objet = Private_Key_Object
                        Type_Clé = Private_Key_RSA
                        Label_Clé = « Clé privée de la carte »
                        ID_Clé = 0001
                      }
nbre_attr_Priv_Key_C = 4
nbre_max_Objet = 1
res = C_FindObjectsInit(session, descriptif_Priv_Key_C, nbre_attr_Priv_Key_C)
res = C_FindObjects(session, Skc, nbre_max_Objet, num_dern_Objet)
```

Chiffrer le nombre aléatoire du prestataire de service

```
res = C_SignRecoverInit(session, Type_Mec, Skc)
res = C_SignRecover(session, Rps, sizeof(Rps), R_Rps, Long_R_Rps)
```

Créer un objet contenant (Idps : l'identifiant et Pkps : la clé publique du prestataire de services)

Rem : Nous allons créer un objet clé publique contenant la clé publique du prestataire de services (Pkps) dont l'identifiant sera l'identifiant du prestataire de services (Idps). Nous extrayons de la donnée Res_Verif l'identifiant, le modulo et l'exposant public permettant de créer la clé.

```
descriptif_Pub_Key_PS = {
                        Type_Objet = Public_Key_Object
                        Type_Clé = Public_Key_RSA
                        Label_Clé = « Clé publique du prestataire de services »
                        ID_Clé = Idps
                        Modulus = {...}
                        Long_Key = 512
                        Pub_exponent = {...}
                      }
nbre_attr_Pub_Key_PS = 7
res = C_CreateObject(session,descriptif_Pub_Key_PS, nbre_attr_Pub_Key_PS,
num_Pub_Key)
```

6.4.1.3 Traitement

Cette partie (Figure 6 - 13) matérialise le travail du prestataire de services (point (6) Figure 6 - 8). Elle a pour but de :

- vérifier la signature du certificat venant de la carte (point (5) de la Figure 6 - 8) à l'aide de la fonction **C_VerifyRecover** (point (6) de la Figure 6 - 8),
- vérifier la signature du nombre aléatoire du prestataire de services (Rps) chiffré par la carte (point (6) de la Figure 6 - 8),
- vérifier que le résultat de ce déchiffrement correspond bien au nombre aléatoire envoyé au point (2) de la Figure 6 - 8,
- signer le nombre aléatoire venant de la carte à l'aide de la fonction **C_SignRecover** (point (7) de la Figure 6 - 8),

Au terme de cette partie, le prestataire de services a pu ou non authentifier la carte.

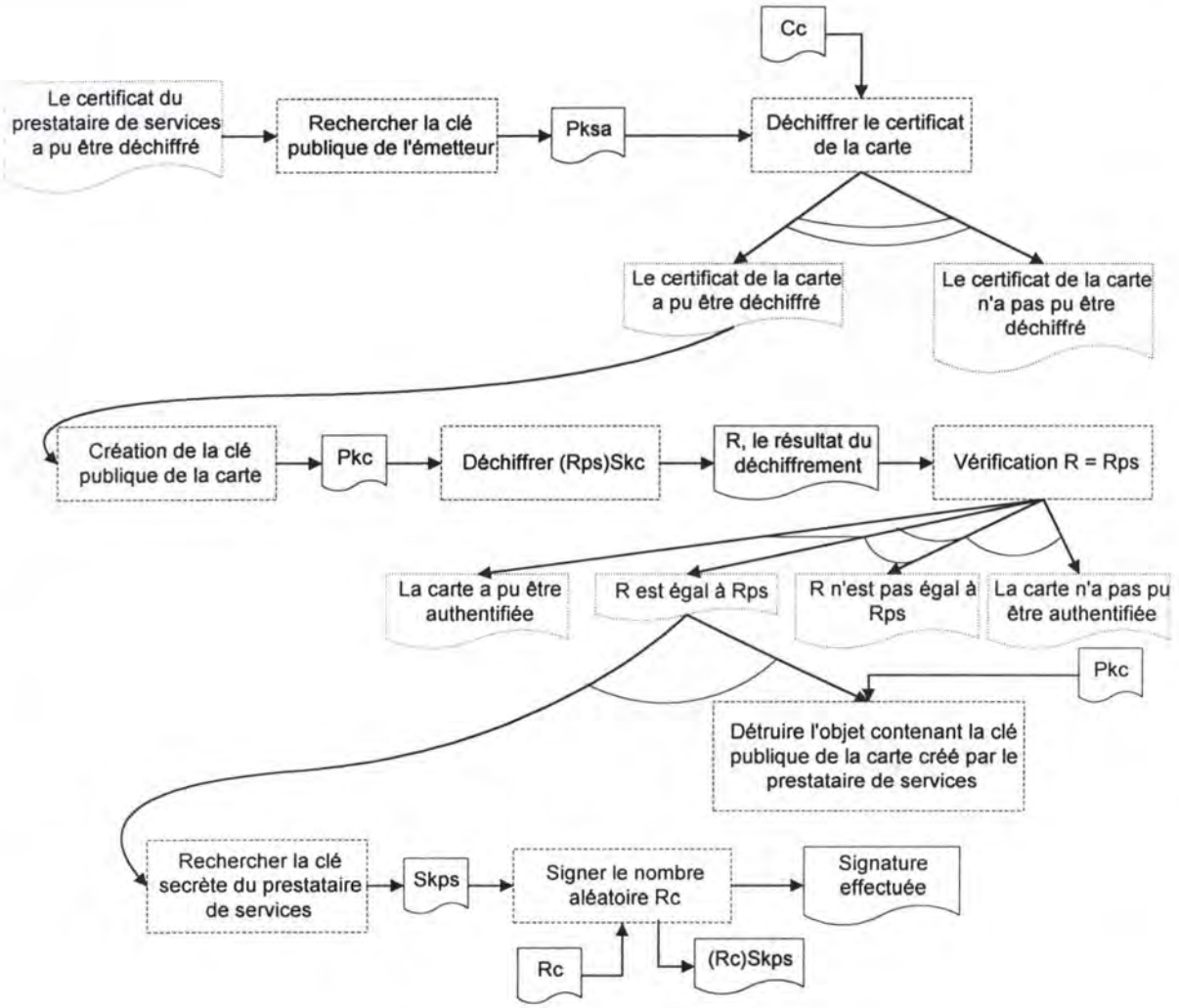


Figure 6 - 13 : Le Traitement.

Soient :

- session = sesPS, le numéro de session du prestataire de services;
- descriptif_Pub_Key_SA, le descriptif de l'objet qui contient la clé publique de l'émetteur :
  - Type_Objet = Public_Key_Object, le type d'objet,
  - Type_Clé = Public_Key_RSA, le type de clé,
  - Label_Clé = « Clé publique de l'émetteur », le label de la clé,
  - ID_Clé = 0010, l'identifiant de la clé;
- nbre_attr_descr_Pub_Key_SA, le nombre d'attributs contenus dans descriptif_Pub_Key_SA;
- nbre_max_Objet, le nombre maximum de numéros d'objets que l'on souhaite recevoir;
- num_dern_Objet, le numéro du dernier objet trouvé;
- Res_Verif, le résultat de la vérification du certificat de la carte (Cc);
- res, l'état final de l'exécution des fonctions **C_FindObjectInit**, **C_FindObjects**, **C_VerifyRecoverInit**, **C_VerifyRecover**, **C_CreateObject**, **C_SignRecoverInit**, **C_SignRecover**;
- Pksa, le numéro de l'objet contenant la clé publique de l'émetteur;
- Pkc, le numéro de l'objet contenant la clé publique de la carte;

- Skps, le numéro de l'objet contenant la clé privée du prestataire de services;
- Type_Mec = PCKS#1_RSA, le type de mécanisme de vérification;
- $Bloc_i$, le $i^{ème}$ bloc du certificat de la carte (Cc);
- Sizeof($Bloc_i$), la longueur du $i^{ème}$ bloc du certificat de la carte;
- R_Verif_i, le résultat de la vérification du $i^{ème}$ bloc;
- $Long\_R\_Verfi_i$, la longueur du résultat de la vérification du $i^{ème}$ bloc;
- i = 1,...,n
- descriptif_Pub_Key_C, le descriptif de la clé publique que l'on désire créer :
  - Type_Objet = Public_Key_Object, le type d'objet,
  - Type_Clé = Public_Key_RSA, le type de clé,
  - Label_Clé = « Clé publique de la carte », le label de la clé,
  - ID_Clé = Idc, l'identifiant de la clé,
  - Modulus, le Modulo *n*,
  - Long_Key = 512, la longueur (en bits) de la clé que l'on veut créer,
  - Pub_exponent, l'exposant public *e*;
- nbre_attr_Pub_Key_C, le nombre d'attributs contenus dans le descriptif de la clé;
- R_Rps, le résultat de la signature du nombre aléatoire du prestataire de services;
- Long_R_Rps, la longueur du résultat R_Rps;
- R, le résultat de la vérification de R_Rps;
- Long_R, la longueur de ce résultat;
- descriptif_Priv_Key_PS, le descriptif de la clé privée que l'on désire trouver :
  - Type_Objet = Private_Key_Object, le type d'objet,
  - Type_Clé = Private_Key_RSA, le type de clé,
  - Label_Clé = « Clé privée du prestataire de services », le label de la clé,
  - ID_Clé = 0001, l'identifiant de la clé;
- nbre_attr_Priv_Key_PS, le nombre d'attributs contenus dans le descriptif de la clé privée;
- Rc, le nombre aléatoire de la carte;
- R_Rc, le résultat de la signature du nombre aléatoire de la carte;
- Long_R_Rc, la longueur du résultat R_Rc.

Le corps de cette partie peut alors s'écrire :

```
session = sesPS
Rechercher la clé publique de l'émetteur
descriptif_Pub_Key_SA = {
                            Type_Objet = Public_Key_Object
                            Type_Clé = Public_Key_RSA
                            Label_Clé = « Clé publique de l'émetteur »
                            ID_Clé = 0010
                        }
nbre_attr_descr_Pub_Key_SA = 4
nbre_max_Objet = 1
res = C_FindObjectInit(session, descriptif_Pub_Key_SA, nbre_attr_descr_Pub_Key_SA)
res = C_FindObjects(session, Pksa, nbre_max_Objet, num_dern_Objet)

Déchiffrer le certificat de la carte
Type_Mec = PCKS#1_RSA
Tant que i <= n
   Bloc_i = {...}
```

```
   res = C_VerifyRecoverInit(session, Type_Mec, Pksa)
   res = C_VerifyRecover(session, Bloc_i, sizeof(Bloc_i), R_Verif_i , Long_R_Verif_i)
   Res_Verif = Res_Verif + R_Verif_i
   i = i +1
Si res <>0
   Alors
      Créer la clé publique de la carte
      Rem : de Res_Verif nous extrayons le modulo et l'exposant public de manière à
      pouvoir construire la clé publique de la carte.
      descriptif_Pub_Key_C = {
                              Type_Objet = Public_Key_Object
                              Type_Clé = Public_Key_RSA
                              Label_Clé = « Clé publique de la carte »
                              ID_Clé = Idc
                              Modulus = {...}
                              Long_Key = 512
                              Pub_exponent = {...}
                            }
      nbre_attr_Pub_Key = 7
      res = C_CreateObject(num_session,descriptif_Pub_Key_C, nbre_attr_Pub_Key_C,
      Pkc)

      Déchiffrer (Rps)Skc et vérifier R = Rps
      res = C_VerifyRecoverInit(session, Type_Mec, Pkc)
      res = C_VerifyRecover(session, R_Rps , Long_R_Rps, R, Long_R)
      Si r = Rcl
         Alors
            Le prestataire de services a authentifié la carte

            Détruire la clé publique de la carte
            res = C_DestroyObject(session, Pkc)

            Rechercher la clé secrète du prestataire de services
            descriptif_Priv_Key_PS = {
                                      Type_Objet = Private_Key_Object
                                      Type_Clé = Private_Key_RSA
                                      Label_Clé = « Clé privée du prestataire de
                                      services »
                                      ID_Clé = 0001
                                    }
            nbre_attr_Priv_Key_PS = 4
            nbre_max_Objet = 1
            res = C_FindObjectInit(session, descriptif_Priv_Key_PS,
            nbre_attr_descr_Priv_Key_PS)
            res = C_FindObjects(session, Skps, nbre_max_Objet, num_dern_Objet)
            Signer le nombre aléatoire de la carte
            res = C_SignRecoverInit(session, Type_Mec, Skps)
            res = C_SignRecover(session, Rc, sizeof(Rc), R_Rc, Long_R_Rc)
```

6.4.1.4 AuthIntCSP

Cette fonction (Figure 6 - 14) matérialise le travail de la carte (point (9) Figure 6 - 8) . Son but est double :

- vérifier la signature du nombre aléatoire de la carte (Rc) chiffré par le prestataire de services,
- vérifier que le résultat de ce déchiffrement correspond bien au nombre aléatoire envoyé au point (5) de la Figure 6 - 8.

Au terme de ces deux vérifications, la carte sera en mesure d'indiquer si elle a authentifié ou non le prestataire de services.

Cette fonction reçoit du prestataire de services un nombre aléatoire signé ((Rc)Skps). Ce qui est représenté par les points (8) de la Figure 6 - 8.

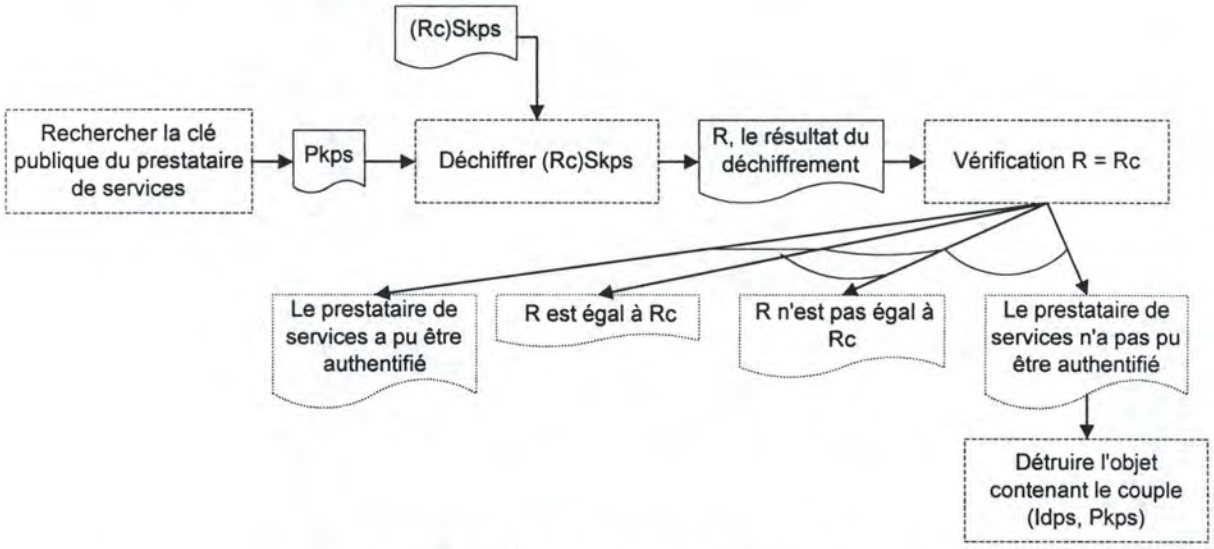


Figure 6 - 14 : Fonction AuthIntCSP.

Soient :

- session = sesCard, le numéro de session de la carte;
- Pkps, le numéro de l'objet contenant la clé publique du prestataire de services;
- descriptif_Pub_Key_PS, le descriptif de l'objet qui contient la clé publique du prestataire de services :
  - Type_Objet = Public_Key_Object, le type d'objet,
  - Type_Clé = Public_Key_RSA, le type de clé,
  - Label_Clé = « Clé publique du prestataire de services », le label de la clé,
  - ID_Clé, l'identifiant de la clé;
- nbre_attr_descr_Pub_Key_PS, le nombre d'attributs contenus dans descriptif_Pub_Key_PS;
- nbre_max_Objet, le nombre maximum de numéros d'objets que l'on souhaite recevoir;
- num_dern_Objet, le numéro du dernier objet trouvé;
- R_Rc, le résultat de la signature du nombre aléatoire de la carte;
- Long_R_Rc, la longueur du résultat R_Rc;

- R, le résultat de la vérification de R_Rc;
- Long_R, la longueur de ce résultat;
- res, l'état final de l'exécution des fonctions **C_FindObjectInit**, **C_FindObjects**, **C_VerifyRecoverInit**, **C_VerifyRecover**, **C_DestroyObject**.

Le corps de cette fonction peut alors s'écrire :

```
session = sesCard
Rechercher la clé publique du prestataire de services
descriptif_Pub_Key_PS = {
                        Type_Objet = Public_Key_Object
                        Type_Clé = Public_Key_RSA
                        Label_Clé = « Clé publique du prestataire de services »
                        ID_Clé = {...}
                    }
nbre_attr_descr_Pub_Key_PS = 4
nbre_max_Objet = 1
res = C_FindObjectInit(session, descriptif_Pub_Key_PS, nbre_attr_descr_Pub_Key_PS)
res = C_FindObjects(session, Pkps, nbre_max_Objet, num_dern_Objet)

Déchiffrer (Rc)Skps et vérifier R = Rc
        res = C_VerifyRecoverInit(session, Type_Mec, Pkps)
        res = C_VerifyRecover(session, R_Rc , Long_R_Rc, R, Long_R)
        Si r = Rc
           Alors
              La carte a authentifié le prestataire de services
           Sinon
              La carte n'a pas authentifié le prestataire de services

              Détruire l'objet contenant la clé publique du prestataire de services
              C_DestroyObjet(session, Pkps)
```

6.4.2 L'authentification Carte - Client

Afin d'expliquer clairement ce que réalise chaque fonction, reprenons la représentation du protocole d'authentification Carte-Client (Figure 6 - 15).

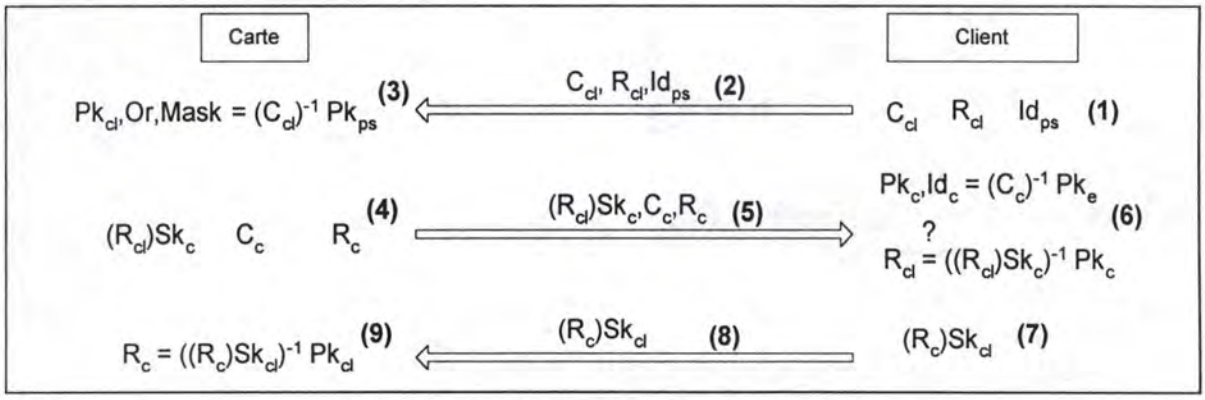


Figure 6 - 15 : Protocole d'authentification Carte - Client.

Cette sous-section a pour but de spécifier la fonction ***AuthentifCCU*** (Figure 6 - 16) qui matérialise l'authentification mutuelle d'une carte et d'un client. Cette fonction va réaliser le protocole de la Figure 6 - 15.

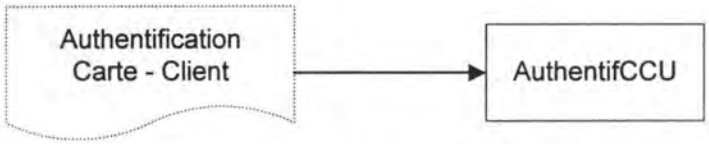


Figure 6 - 16 : Authentification Carte - Client.

Cette fonction se découpe en quatre parties (Figure 6 - 17) que nous spécifions ci-dessous.

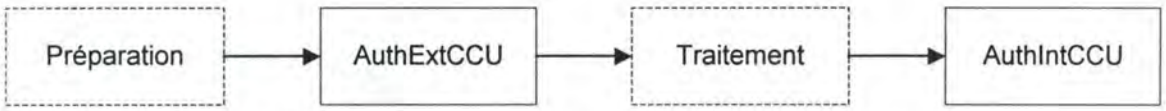


Figure 6 - 17 : Décomposition de la fonction AuthentifCCU.

6.4.2.1 La préparation

La tâche (Figure 6 - 18), reprise sous le nom de *Préparation* (appartenant à la fonction ***AuthentifCCU***), matérialise le travail du client. Elle a pour but de générer un nombre aléatoire à l'aide de la fonction **C_GenerateRandom** et de rechercher parmi tous les objets du client celui qui contient son certificat à l'aide des fonctions **C_FindObjets** et **C_GetAttributeValue**. Vu qu'un client peut disposer de plusieurs certificats provenant de prestataires de services différents, nous avons créé une table de correspondance gérée par le client et reprenant les couples (nom du prestataire de services, numéro de l'objet contenant le certificat émis par ce prestataire). Une recherche à partir du nom du prestataire de services permet de retrouver le numéro de l'objet contenant le certificat du client.

Nous disposons également d'une table reprenant les couples (nom du prestataire de services, numéro de l'objet contenant l'identifiant de ce prestataire) nous permettant de retrouver l'identifiant du prestataire qui a émis le certificat du client.

Elle réalise en réalité le point (1) du protocole (Figure 6 - 15). La Figure 6 - 18 représente les tâches réalisées par la *Préparation*.

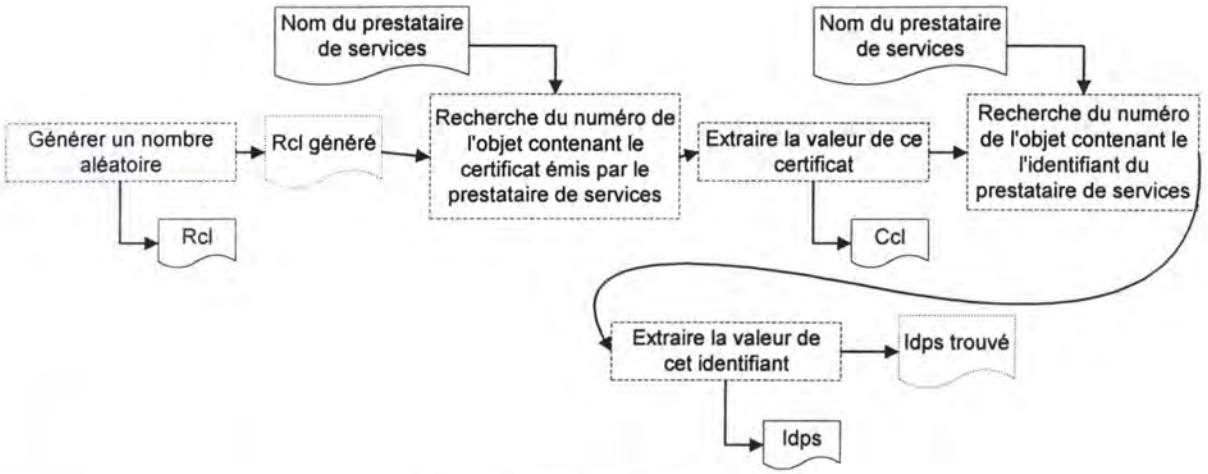


Figure 6 - 18 : La Préparation.

Rcl est le nombre aléatoire du client, *Ccl* étant son certificat.

Soient :

- session = sesCU, le numéro de la session du client;
- Table_Certif, la table de correspondance reprenant le nom du prestataire de services et le numéro de l'objet contenant le certificat émis par ce prestataire;
- Table_Id, la table de correspondance reprenant le nom du prestataire de services et le numéro de l'objet contenant l'identifiant de ce prestataire;
- Rcl, le nombre aléatoire généré par le client;
- num_certif, le numéro de l'objet contenant le certificat du client;
- nom_prestataire, le nom du prestataire de services qui a émis le certificat pour ce client;
- res, l'état final de l'exécution des fonctions **C_GetAttributeValue** et **C_GenerateRandom**;
- descriptif_Certif_Info, le descriptif des attributs (du certificat) pour lesquels on souhaite une valeur :
  - Ccl, le certificat trouvé;
- nbre_attr_descriptif_Certif_Info, le nombre d'attributs contenus dans descriptif_Certif_Info;
- descriptif_Id_Info, le descriptif des attributs (du certificat) pour lesquels on souhaite une valeur :
  - Idps, l'identifiant trouvé;
- nbre_attr_descriptif_Id_Info, le nombre d'attributs contenus dans descriptif_Id_Info.

Le corps de la partie de code de la fonction ***AuthentifCCU*** reprise sous le nom *Préparation* peut alors s'écrire :

```
session = sesCU
Génération du nombre aléatoire du client
res = C_GenerateRandom(session, Rcl, sizeof(Rcl))

Recherche dans la table de correspondance, à partir du nom du prestataire de services, du
numéro de l'objet contenant le certificat du client émis par ce prestataire

num_certif = Recherche(Table_Certif, nom_prestataire)

Extraction de la valeur du certificat
descriptif_Certif_Info = {
                              Ccl
                         }
nbre_attr_descriptif_Certif_Info = 1
res = C_GetAttributeValue(session, num_certif, descriptif_Certif_Info,
nbre_attr_descriptif_Certif_Info)

Recherche dans la table de correspondance, à partir du nom du prestataire de services, du
numéro de l'objet contenant l'identifiant de ce prestataire
num_id = Recherche(Table_Id, nom_prestataire)

Extraction de la valeur de l'identifiant
descriptif_Id_Info = {
                              Idps
                         }
nbre_attr_descriptif_Id_Info = 1
res = C_GetAttributeValue(session, num_id, descriptif_Id_Info, nbre_attr_descriptif_Id_Info)
```

6.4.2.2 AuthExtCCU

La fonction AuthExtCCU (dont le corps est représenté à la Figure 6 - 19) matérialisant le travail de la carte, a pour but de :

- vérifier la signature du certificat venant du client à l'aide de la fonction **C_VerifyRecover** (point (3) de la Figure 6 - 15),
- rechercher parmi tous ses objets celui qui contient son certificat à l'aide des fonctions **C_FindObjects** et **C_GetAttributeValue** (point (4) de la Figure 6 - 15),
- générer son nombre aléatoire à l'aide de la fonction **C_GenerateRandom** (point (4) de la Figure 6 - 15),
- signer le nombre aléatoire venant du client à l'aide de la fonction **C_SignRecover** (point (4) de la Figure 6 - 15),
- créer un objet contenant la clé publique du client.

Cette fonction reçoit le certificat *Ccl*, le nombre aléatoire *Rcl* et l'identifiant du prestataire de services (qui a émis le certificat) et renvoie le certificat *Cc*, le nombre aléatoire de la carte et le nombre aléatoire du client signé. Ce qui est représenté par les points (2) et (5) de la Figure 6 - 15.

Le certificat *Ccl* sera décomposé en blocs afin de respecter les contraintes de taille de données lors d'une vérification de signature.

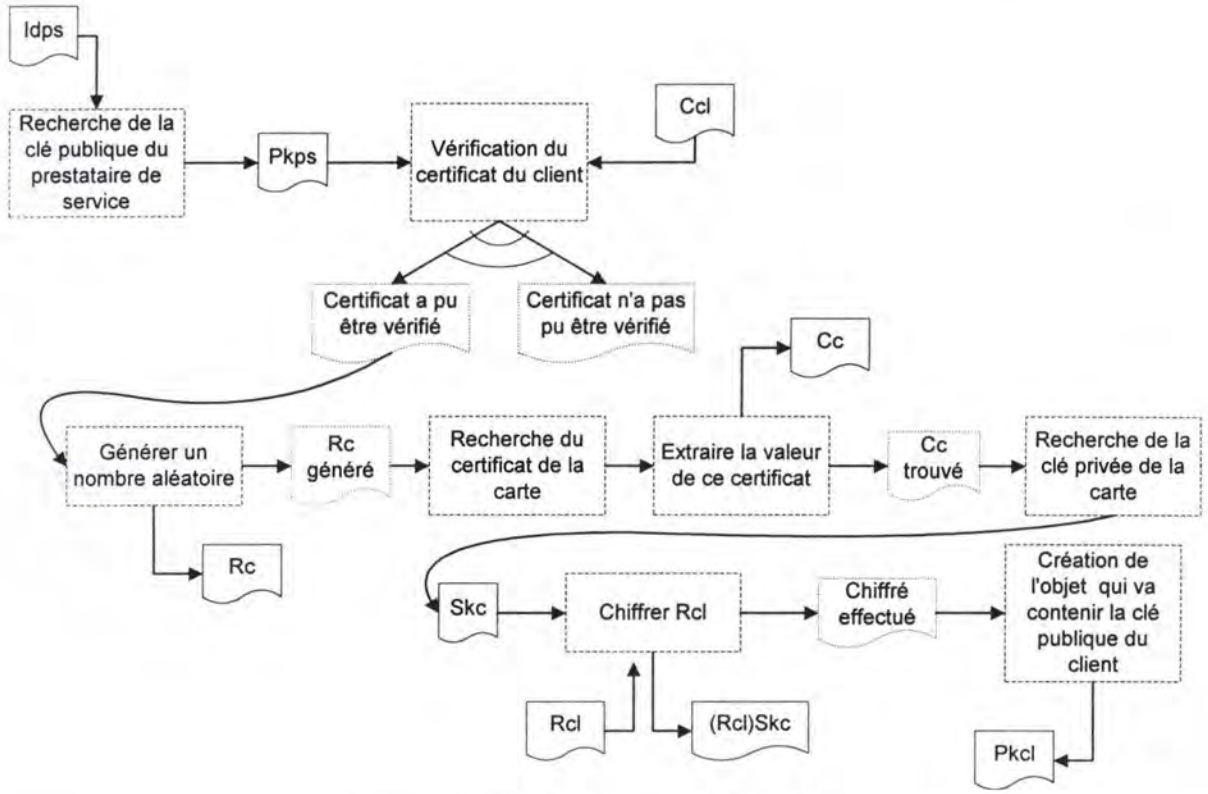


Figure 6 - 19 : Fonction AuthExtCCU.

Soient

- Idps, l'identifiant du prestataire de services;
- Pkps, le numéro de l'objet contenant la clé publique du prestataire de services;
- Skc, le numéro de l'objet contenant la clé privée de la carte;
- Rc, le nombre aléatoire de la carte;
- Rcl, le nombre aléatoire du client;
- Ccl, le certificat du client;
- Certif_C, le numéro de l'objet contenant le certificat de la carte;
- session = sesCard, le numéro de session de la carte;
- descriptif_Pub_Key_PS, le descriptif de l'objet qui contient la clé publique du prestataire de services :
  - Type_Objet = Public_Key_Object, le type d'objet,
  - Type_Clé = Public_Key_RSA, le type de clé,
  - Label_Clé = « Clé publique du prestataire de services », le label de la clé,
  - ID_Clé, l'identifiant de la clé;
- nbre_attr_descr_Pub_Key_PS, le nombre d'attributs contenus dans descriptif_Pub_Key_PS;
- nbre_max_Objet, le nombre maximum de numéros d'objets que l'on souhaite recevoir;
- num_dern_Objet, le numéro du dernier objet trouvé;
- Res_Verif, le résultat de la vérification du certificat du client;
- res, l'état final de l'exécution des fonctions **C_FindObjectInit**, **C_FindObjects**, **C_GetAttributeValue**, **C_GenerateRandom**, **C_VerifyRecoverInit**, **C_VerifyRecover**, **C_SignRecoverInit** et **C_SignRecover**;
- Type_Mec = PCKS#1_RSA, le type de mécanisme de vérification;

- $Bloc_i$, le $i^{ème}$ bloc du certificat du client (Ccu);
- $Sizeof(Bloc_i)$, la longueur du $i^{ème}$ bloc du certificat du client;
- R_Verif_i, le résultat de la vérification du $i^{ème}$ bloc;
- $Long\_R\_Verfi_i$, la longueur du résultat de la vérification du $i^{ème}$ bloc;
- i = 1,...,n
- descriptif_certificat, le descriptif de l'objet qui contient le certificat de la carte :
  - Type_Objet = Data_Object, le type de l'objet,
  - Label_Objet = « certificat de la carte », le label de l'objet,
  - carac_objet = public, le caractère de l'objet,
  - appli = « application », le label de l'application qui gère l'objet contenant le certificat;
- nbre_attr_descriptif_certificat, le nombre d'attributs de ce descriptif;
- descriptif_Certif_Info, le descriptif des attributs (du certificat) pour lesquels on souhaite une valeur :
  - Cc, le certificat trouvé;
- nbre_attr_descriptif_Certif_Info, le nombre d'attributs contenus dans descriptif_Certif_Info;
- descriptif_Priv_Key_C, le descriptif de la clé privée (de la carte) que l'on recherche:
  - Type_Objet = Private_Key_Object, le type d'objet,
  - Type_Clé = Private_Key_RSA, le type de clé,
  - Label_Clé = « Clé privée de la carte », le label de la clé,
  - ID_Clé = 0001, l'identifiant de la clé;
- nbre_attr_Priv_Key_C, le nombre d'attributs contenus dans le descriptif de la clé privée;
- R_Rcl, le résultat de la signature du nombre aléatoire du client;
- Long_R_Rcl, la longueur du résultat R_Rcl;
- descriptif_Pub_Key_CL, le descriptif de la clé publique que l'on désire créer :
  - Type_Objet = Public_Key_Object, le type d'objet,
  - Type_Clé = Public_Key_RSA, le type de clé,
  - Label_Clé = « Clé publique du client », le label de la clé,
  - ID_Clé, l'identifiant de la clé,
  - Modulus, le Modulo *n*,
  - Long_Key = 512, la longueur (en bits) de la clé que l'on veut créer,
  - Pub_exponent, l'exposant public *e*;
- nbre_attr_Pub_Key_CL, le nombre d'attributs contenus dans le descriptif de la clé;
- Pkcl, le numéro de l'objet qui va contenir (dans la carte) la clé publique du client.

Le corps de la fonction peut alors s'écrire :

```
session = sesCard
Recherche de la clé publique du prestataire de services
descriptif_Pub_Key_PS = {
                                Type_Objet = Public_Key_Object
                                Type_Clé = Public_Key_RSA
                                Label_Clé = « Clé publique du prestataire de services »
                                ID_Clé = {...}
                        }
```

```
nbre_attr_descr_Pub_Key_PS = 4
nbre_max_Objet = 1
res = C_FindObjectInit(session, descriptif_Pub_Key_PS, nbre_attr_descr_Pub_Key_PS)
res = C_FindObjects(session, Pkps, nbre_max_Objet, num_dern_Objet)

Vérification du certificat du client
Type_Mec = PCKS#1_RSA
Tant que i <= n
   Bloc_i = {...}
   res = C_VerifyRecoverInit(session, Type_Mec, Pkps)
   res = C_VerifyRecover(session, Bloc_i, sizeof(Bloc_i), R_Verif_i, Long_R_Verif_i)
   Res_Verif = Res_Verif + R_Verif_i
   i = i +1
Si res <>0
   Alors
      Le certificat du client a pu être vérifié

      Générer le nombre aléatoire de la carte
      res = C_GenerateRandom(session, Rc, sizeof(Rc))

      Recherche de l'objet contenant le certificat de la carte
      descriptif_certificat = {
                              Type_Objet = Data_Object,
                              Label_Objet = « certificat de la carte »,
                              carac_objet = public,
                              appli
                           }
      nbre_attr_descriptif_certificat = 4
      nbre_max_Objet = 1
      res = C_FindObjectsInit(session, descriptif_certificat, nbre_attr_descriptif_certificat)
      res = C_FindObjects(session, Certif_C, nbre_max_Objet, num_dern_Objet)

      Extraction de la valeur du certificat
      descriptif_Certif_Info = {
                                 Cc
                              }

      nbre_attr_descriptif_Certif_Info = 1
      res = C_GetAttributeValue(session, Certif_C, descriptif_Certif_Info,
      nbre_attr_descriptif_Certif_Info)

      Rechercher la clé privée de la carte
      descriptif_Priv_Key_C = {
                              Type_Objet = Private_Key_Object
                              Type_Clé = Private_Key_RSA
                              Label_Clé = « Clé privée de la carte »
                              ID_Clé = 0001
                           }
      nbre_attr_Priv_Key_C = 4
      nbre_max_Objet = 1
      res = C_FindObjectsInit(session, descriptif_Priv_Key_C, nbre_attr_Priv_Key_C)
      res = C_FindObjects(session, Skc, nbre_max_Objet, num_dern_Objet)
```

Chiffrer le nombre aléatoire du client
res = **C_SignRecoverInit**(session, Type_Mec, Skc)
res = **C_SignRecover**(session, Rcl, sizeof(Rcl), R_Rcl, Long_R_Rcl)

Créer un objet contenant la clé publique du client

Rem : Nous extrayons de la donnée Res_Verif le modulo et l'exposant public permettant de créer la clé du client.

```
descriptif_Pub_Key_CL = {
                    Type_Objet = Public_Key_Object
                    Type_Clé = Public_Key_RSA
                    Label_Clé = « Clé publique du client »
                    ID_Clé = {...}
                    Modulus = {...}
                    Long_Key = 512
                    Pub_exponent = {...}
                }
nbre_attr_Pub_Key_CL = 7
res = C_CreateObject(session,descriptif_Pub_Key_CL, nbre_attr_Pub_Key_CL, Pkcl)
```

6.4.2.3 Traitement

Cette partie (Figure 6 - 20) matérialise le travail du client (point (6) Figure 6 - 15). Elle a pour but de :

- vérifier la signature du certificat venant de la carte (point (5) Figure 6 - 15) à l'aide de la fonction **C_VerifyRecover** (point (6) Figure 6 - 15),
- vérifier la signature du nombre aléatoire du client (Rcl) chiffré par la carte (point (6) Figure 6 - 15),
- vérifier que le résultat de ce déchiffrement correspond bien au nombre aléatoire envoyé au point (2) de la Figure 6 - 15,
- signer le nombre aléatoire venant de la carte à l'aide de la fonction **C_SignRecover** (point (7) de la Figure 6 - 15),

Au terme de cette partie le client a pu ou non authentifier la carte.

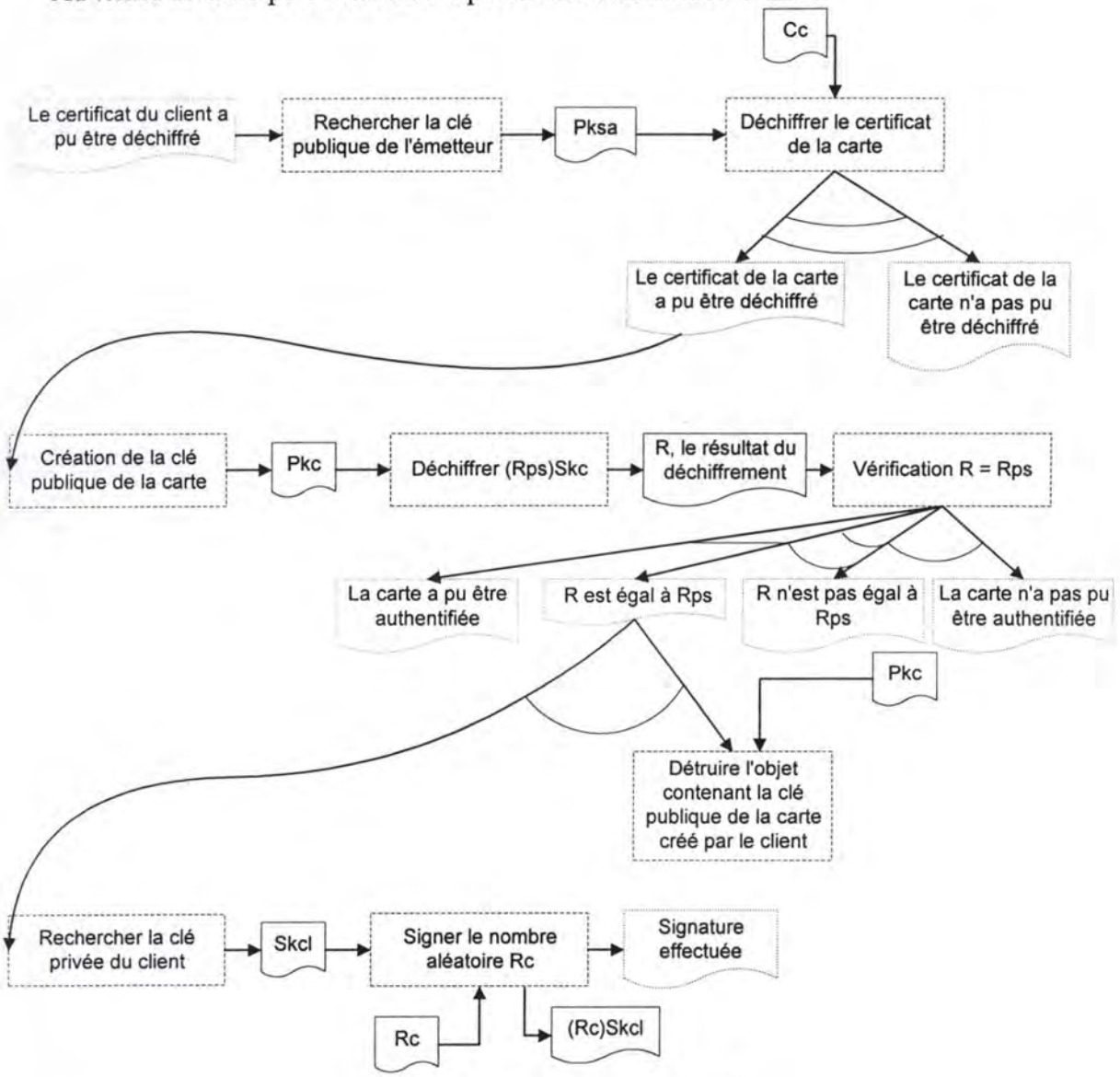


Figure 6 - 20 : Le Traitement.

Soient :

- session = sesCU, le numéro de session du client;
- descriptif_Pub_Key_SA, le descriptif de l'objet qui contient la clé publique de l'émetteur :
  - Type_Objet = Public_Key_Object, le type d'objet,
  - Type_Clé = Public_Key_RSA, le type de clé,
  - Label_Clé = « Clé publique de l'émetteur », le label de la clé,
  - ID_Clé = 0010, l'identifiant de la clé;
- nbre_attr_descr_Pub_Key_SA, le nombre d'attributs contenus dans descriptif_Pub_Key_SA;
- nbre_max_Objet, le nombre maximum de numéros d'objets que l'on souhaite recevoir;
- num_dern_Objet, le numéro du dernier objet trouvé;
- Res_Verif, le résultat de la vérification du certificat de la carte;

- res, l'état final de l'exécution des fonctions **C_FindObjectInit**, **C_FindObjects**, **C_VerifyRecoverInit**, **C_VerifyRecover**, **C_CreateObject**, **C_SignRecoverInit**, **C_SignRecover**;
- Pksa, le numéro de l'objet contenant la clé publique de l'émetteur;
- Pkc, le numéro de l'objet contenant la clé publique de la carte;
- Skcl, le numéro de l'objet contenant la clé privée du client;
- Type_Mec = PCKS#1_RSA, le type de mécanisme de vérification;
- $Bloc_i$, le $i^{ème}$ bloc du certificat de la carte (Cc);
- $Sizeof(Bloc_i)$, la longueur du $i^{ème}$ bloc du certificat de la carte;
- R_Verif_i, le résultat de la vérification du $i^{ème}$ bloc;
- $Long\_R\_Verfi_i$, la longueur du résultat de la vérification du $i^{ème}$ bloc;
- i = 1,...,n
- descriptif_Pub_Key_C, le descriptif de la clé publique (de la carte) que l'on désire créer :
  - Type_Objet = Public_Key_Object, le type d'objet,
  - Type_Clé = Public_Key_RSA, le type de clé,
  - Label_Clé = « Clé publique de la carte », le label de la clé,
  - ID_Clé = Idc, l'identifiant de la clé,
  - Modulus, le Modulo *n*,
  - Long_Key = 512, la longueur (en bits) de la clé que l'on veut créer,
  - Pub_exponent, l'exposant public *e*;
- nbre_attr_Pub_Key_C, le nombre d'attributs contenus dans le descriptif de la clé;
- R_Rcl, le résultat de la signature du nombre aléatoire du client;
- Long_R_Rcl, la longueur du résultat R_Rps;
- R, le résultat de la vérification de R_Rcl;
- Long_R, la longueur de ce résultat;
- descriptif_Priv_Key_CL, le descriptif de la clé privée (du client) que l'on désire trouver :
  - Type_Objet = Private_Key_Object, le type d'objet,
  - Type_Clé = Private_Key_RSA, le type de clé,
  - Label_Clé = « Clé privée du client », le label de la clé,
  - ID_Clé, l'identifiant de la clé;
- nbre_attr_Priv_Key_CL, le nombre d'attributs contenus dans le descriptif de la clé privée;
- Rc, le nombre aléatoire de la carte;
- R_Rc, le résultat de la signature du nombre aléatoire de la carte;
- Long_R_Rc, la longueur du résultat R_Rc.

Le corps de cette partie peut alors s'écrire :

```
session = sesCU
Rechercher la clé publique de l'émetteur
descriptif_Pub_Key_SA = {
                        Type_Objet = Public_Key_Object
                        Type_Clé = Public_Key_RSA
                        Label_Clé = « Clé publique de l'émetteur »
                        ID_Clé = 0010
                    }
nbre_attr_descr_Pub_Key_SA = 4
```

```
nbre_max_Objet = 1
res = C_FindObjectInit(session, descriptif_Pub_Key_SA, nbre_attr_descr_Pub_Key_SA)
res = C_FindObjects(session, Pksa, nbre_max_Objet, num_dern_Objet)
Déchiffrer le certificat de la carte
Type_Mec = PCKS#1_RSA
Tant que i <= n
   Bloc_i = {...}
   res = C_VerifyRecoverInit(session, Type_Mec, Pksa)
   res = C_VerifyRecover(session, Bloc_i, sizeof(Bloc_i), R_Verif_i, Long_R_Verif_i)
   Res_Verif = Res_Verif + R_Verif_i
   i = i +1
Si res <>0
   Alors
      Créer la clé publique de la carte
      Rem : de Res_Verif nous extrayons le modulo et l'exposant public de manière à
      pouvoir construire la clé de la carte.
      descriptif_Pub_Key_C = {
                              Type_Objet = Public_Key_Object
                              Type_Clé = Public_Key_RSA
                              Label_Clé = « Clé publique de la carte »
                              ID_Clé = Idc
                              Modulus = {...}
                              Long_Key = 512
                              Pub_exponent = {...}
                            }
      nbre_attr_Pub_Key = 7
      res = C_CreateObject(num_session,descriptif_Pub_Key_C, nbre_attr_Pub_Key_C,
      Pkc)
      Déchiffrer (Rcl)Skc et vérifier R = Rcl
      res = C_VerifyRecoverInit(session, Type_Mec, Pkc)
      res = C_VerifyRecover(session, R_Rcl , Long_R_Rcl, R, Long_R)
      Si r = Rps
         Alors
            Le client  a authentifié la carte

            Détruire l'objet contenant la clé publique de la carte
            res = C_DestroyObjet(session, Pkc)

            Rechercher la clé privée du client
            descriptif_Priv_Key_CL = {
                                      Type_Objet = Private_Key_Object
                                      Type_Clé = Private_Key_RSA
                                      Label_Clé = « Clé privée du prestataire de
                                      services »
                                      ID_Clé = {...}
                                    }
            nbre_attr_Priv_Key_CL = 4
            nbre_max_Objet = 1
            res = C_FindObjectInit(session, descriptif_Priv_Key_CL,
            nbre_attr_descr_Priv_Key_CL)
            res = C_FindObjects(session, Skcl, nbre_max_Objet, num_dern_Objet)
            Signer le nombre aléatoire de la carte
            res = C_SignRecoverInit(session, Type_Mec, Skcl)
```

res = **C_SignRecover**(session, Rc, sizeof(Rc), R_Rc, Long_R_Rc)

6.4.2.4 AuthIntCCU

Cette fonction (Figure 6 - 21) matérialise le travail de la carte (point (9) Figure 6 - 15) . Son but est double :

- vérifier la signature du nombre aléatoire de la carte (Rc) chiffré par le client,
- vérifier que le résultat de ce déchiffrement correspond bien au nombre aléatoire envoyé au point (5) de la Figure 6 - 15.

Au terme de ces deux vérifications, la carte sera en mesure d'indiquer si elle a authentifié ou non le client.

Cette fonction reçoit du client un nombre aléatoire signé ((Rc)Skcl). Ce qui est représenté par le point (8) de la Figure 6 - 15.

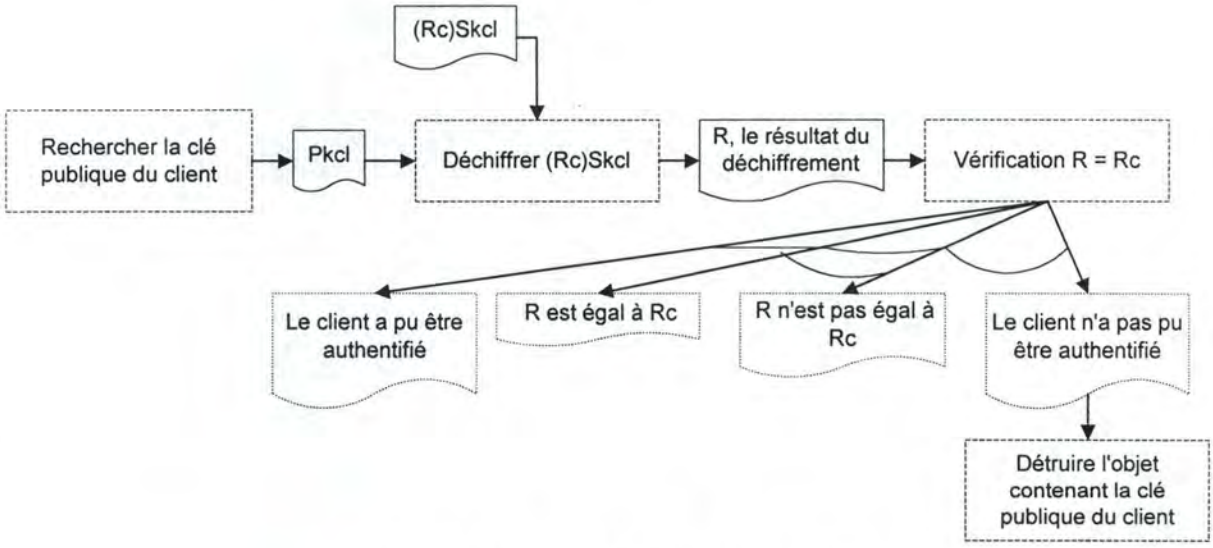


Figure 6 - 21 : Fonction AuthIntCCU.

Soient :

- session = sesCard, le numéro de session de la carte;
- Pkcl, le numéro de l'objet contenant la clé publique du client;
- descriptif_Pub_Key_CL, le descriptif de l'objet qui contient la clé publique du client :
  - Type_Objet = Public_Key_Object, le type d'objet,
  - Type_Clé = Public_Key_RSA, le type de clé,
  - Label_Clé = « Clé publique du client », le label de la clé,
  - ID_Clé, l'identifiant de la clé;
- nbre_attr_descr_Pub_Key_CL, le nombre d'attributs contenus dans descriptif_Pub_Key_CL;
- nbre_max_Objet, le nombre maximum de numéros d'objets que l'on souhaite recevoir;
- num_dern_Objet, le numéro du dernier objet trouvé;
- R_Rc, le résultat de la signature du nombre aléatoire de la carte;

- Long_R_Rc, la longueur du résultat R_Rc;
- R, le résultat de la vérification de R_Rc;
- Long_R, la longueur de ce résultat;
- res, l'état final de l'exécution des fonctions **C_FindObjectInit**, **C_FindObjects**, **C_VerifyRecoverInit**, **C_VerifyRecover**, **C_DestroyObject**.

Le corps de cette fonction peut alors s'écrire :

```
session = sesCard
Rechercher la clé publique du client
descriptif_Pub_Key_CL = {
                        Type_Objet = Public_Key_Object
                        Type_Clé = Public_Key_RSA
                        Label_Clé = « Clé publique du client »
                        ID_Clé = {...}
                    }
nbre_attr_descr_Pub_Key_CL = 4
nbre_max_Objet = 1
res = C_FindObjectInit(session, descriptif_Pub_Key_CL, nbre_attr_descr_Pub_Key_CL)
res = C_FindObjects(session, Pkps, nbre_max_Objet, num_dern_Objet)

Déchiffrer (Rc)Skcl et vérifier R = Rc
        res = C_VerifyRecoverInit(session, Type_Mec, Pkcl)
        res = C_VerifyRecover(session, R_Rc , Long_R_Rc, R, Long_R)
        Si r = Rc
            Alors
                La carte a authentifié le client
            Sinon
                La carte n'a pas authentifié le client

                Détruire l'objet contenant la clé publique du client
                res = C_DestroyObjet(session, Pkcl)
```

Partie 3 : Les Tests et Exemple

Chapitre 7 : Tests

Sommaire

Ce chapitre est consacré à différents tests effectués sur l'application. Ils permettent de montrer, outre que l'application fonctionne, la résistance du protocole à des attaques possibles. Il débute par la présentation de conventions qui faciliteront la compréhension des schémas illustrant les tests effectués. Les résultats des tests seront présentés sous la forme de tableaux et accompagnés de commentaires apportant des précisions sur les résultats.

7.1 Conventions

Il existe deux types de relations entre les partenaires. D'une part la certification et d'autre part l'authentification.

1) A non certifié → A certifié : indique que A passe de l'état non certifié à l'état certifié.
A → B : indique que A donne un certificat à B.
Dans ce cas, A est un prestataire de services ou un émetteur :

- Si A est un prestataire de services, B est obligatoirement un client.
- Si A est un émetteur, B est soit une carte, soit un prestataire de services dont le nom est inscrit sur la flèche.

2) A ↔ B : indique que A et B s'authentifient mutuellement.
Dans ce cas, A est soit une carte, soit un prestataire de services, soit un client.

- Si A est une carte, B est soit un prestataire de services, soit un client.
  Si A est une carte et B un client, alors le nom du prestataire de services qui a certifié le client est inscrit sur la flèche.
- Si A est un prestataire de services, B est obligatoirement une carte.

7.2 Tests du protocole

Ces tests (Figure 7 - 1 & Figure 7 - 2) vont permettre de montrer que le protocole est réalisable à l'aide de fonctions cryptographiques.

7.2.1 Test 1

7.2.1.1 Graphe des interactions

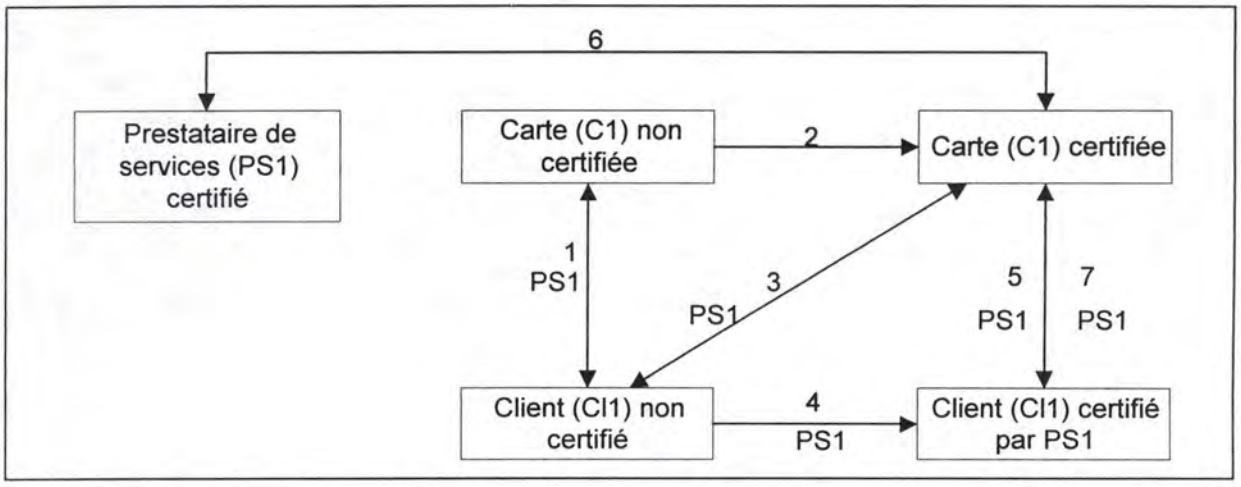


Figure 7 - 1 : Test 1 du protocole.

7.2.1.2 Résultats

Etapes	Carte	Prestataire de services	Client	Emetteur	Type de relation	Résultats	Raisons du résultats
1	C1		CI1		Authentification	Echec	CI1 ne dispose pas du certificat lui permettant de se faire authentifier par C1
2	C1			Emetteur	Certification	Réussite	
3	C1		CI1		Authentification	Echec	CI1 ne dispose pas du certificat lui permettant de se faire authentifier par C1
4			CI1	PS1	Certification	Réussite	
5	C1		CI1		Certification	Echec	C1 n'a pas pu vérifier le certificat de CI1 car PS1 ne s'est pas fait authentifier par C1
6	C1	PS1			Authentification	Réussite	
7	C1		CI1		Authentification	Réussite	

Tableau 7 - 1 : Résultat du test 1.

7.2.1.3 Commentaires

Nous pouvons remarquer que (Tableau 7 - 1) :

- Un client non accrédité ne peut exécuter de services dans la carte.
- Un client dont le prestataire de services qui l'a authentifié n'a chargé aucun service dans la carte, n'a pas accès aux autres services de la carte.

7.2.2 Test 2

7.2.2.1 Graphe des interactions

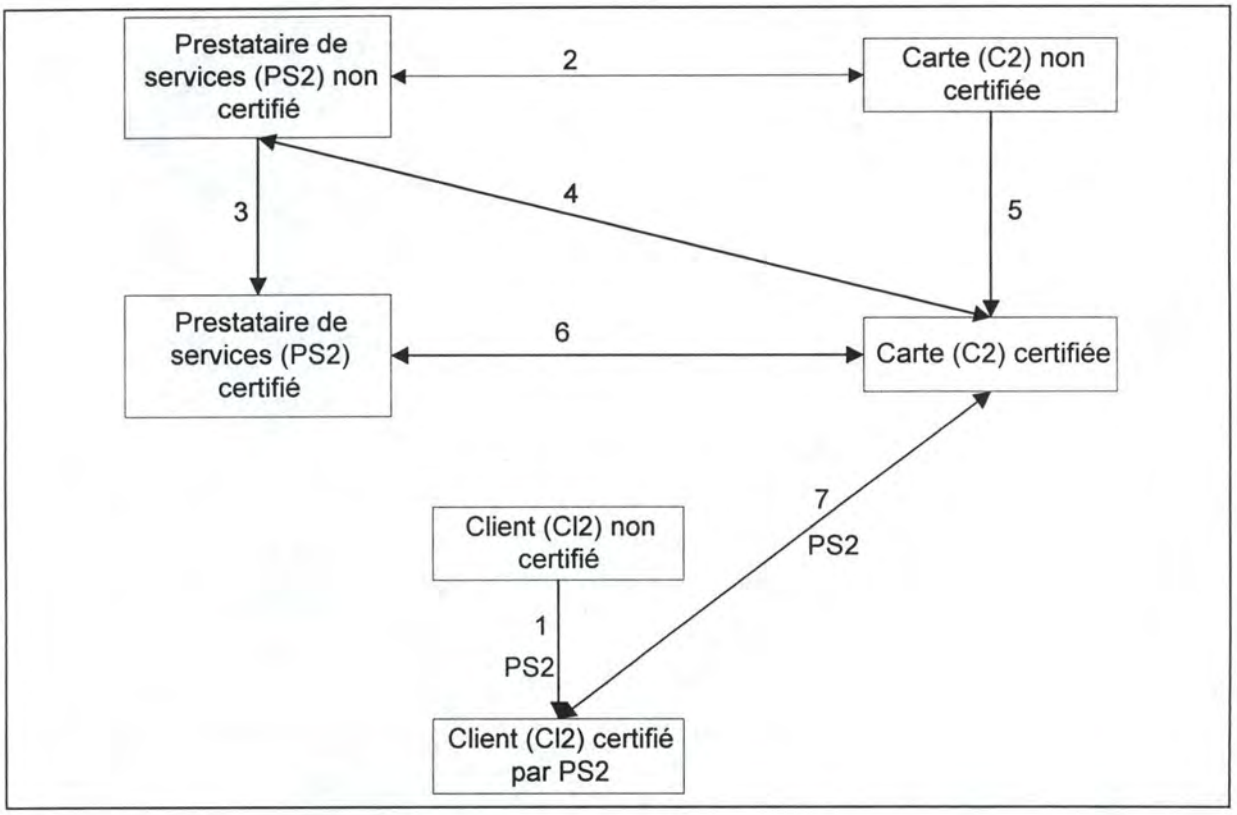


Figure 7 - 2 : Test 2 du protocole.

7.2.2.2 Résultats

Etapes	Carte	Prestataire de services	Client	Emetteur	Type de relation	Résultats	Raisons du résultats
1			CI2	PS2	Certification	Réussite	
2	C2	PS2			Authentification	Echec	PS2 ne dispose pas du certificat lui permettant de se faire authentifier par C2
3		PS2		Emetteur	Certification	Réussite	
4	C2	PS2			Authentification	Echec	C2 ne dispose pas du certificat lui permettant de se faire authentifier par PS2
5	C2			Emetteur	Certifiation	Réussite	
6	C2	PS2			Authentification	Réussite	
7	C2		CI2		Authentification	Réussite	

Tableau 7 - 2 : Résultat du test 2.

7.2.2.3 Commentaires

Nous pouvons remarquer que (Tableau 7 - 2):

- Un prestataire de services non accrédité ne peut charger de services dans une carte.
- Un prestataire de services accrédité ne charge aucun service dans une carte non accréditée.

7.3 Test en grandeur nature

Ce test (Figure 7 - 3) va permettre de montrer que le protocole fonctionne toujours lorsque nous augmentons le nombre de partenaires en présence.

7.3.1 Graphe des interactions

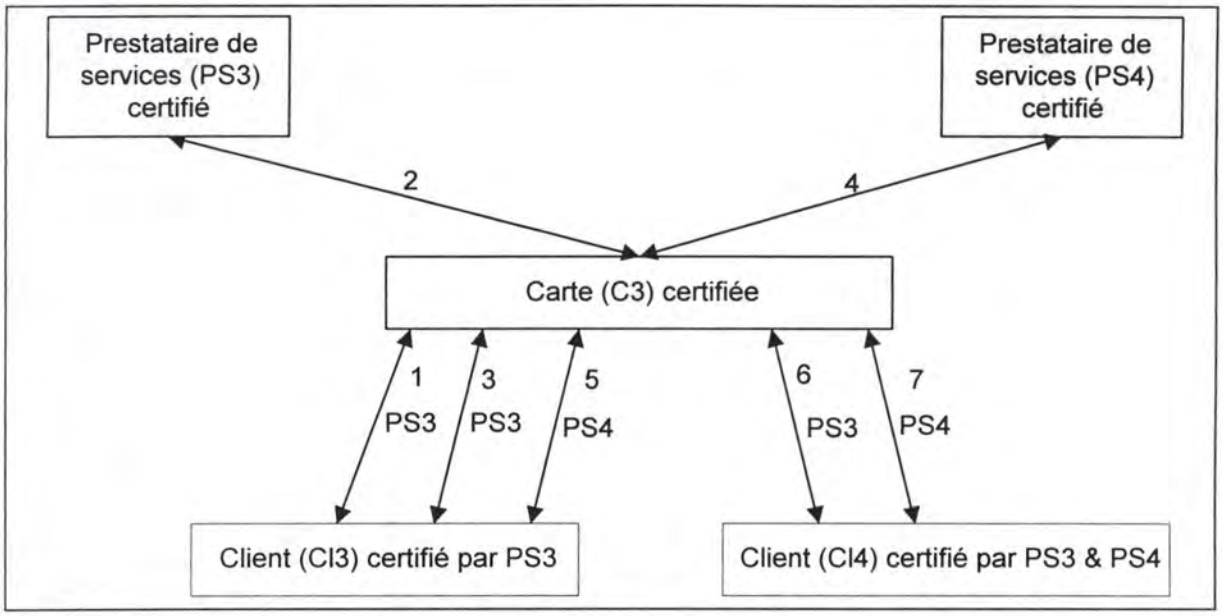


Figure 7 - 3 : Test du protocole en grandeur nature.

7.3.2 Résultats

Etapes	Carte	Prestataire de services	Client	Emetteur	Type de relation	Résultats	Raisons du résultats
1	C3		CI3		Certification	Echec	CI3 ne dispose pas du certificat lui permettant de se faire authentifier par C3
2	C3	PS3			Authentification	Réussite	
3	C3		CI3		Authentification	Réussite	
4	C3	PS4			Authentification	Réussite	
5	C3		CI3		Authentification	Echec	CI3 ne dispose pas du certificat lui permettant de se faire authentifier par C3
6	C3		CI4		Authentification	Réussite	
7	C3		CI4		Authentification	Réussite	

Tableau 7 - 3 : Résultat du test en grandeur nature.

7.3.3 Commentaires

Nous pouvons remarquer que (Tableau 7 - 3):

- Un client accrédité par un prestataire de services ne peut utiliser les services d'un prestataire de services duquel il n'a pas reçu de certificat.
- Un client non accrédité ne peut exécuter de services dans la carte.

7.4 Test d'un cas concret

Ce test (Figure 7 - 4) illustre le protocole dans un cas que l'on peut rencontrer tous les jours.

7.4.1 Graphe des interactions

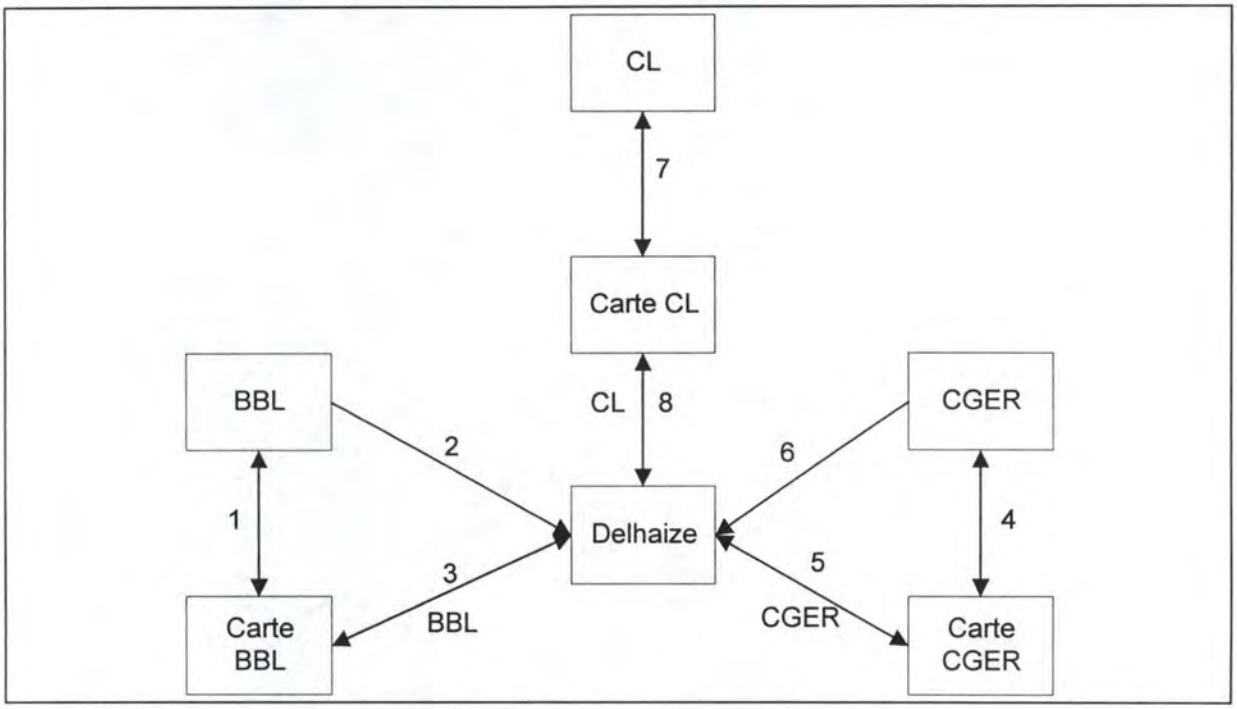


Figure 7 - 4 : Test d'un cas concret.

7.4.2 Résultats

Etapes	Carte	Prestataire de services	Client	Emetteur	Type de relation	Résultats	Raisons du résultats
1	Carte BBL	BBL			Authentification	Réussite	
2		BBL	Delhaize		Certification	Réussite	
3	Carte BBL		Delhaize		Authentification	Réussite	
4	Carte CGER	CGER			Authentification	Réussite	
5	Carte CGER		Delhaize		Authentification	Réussite	
6		CGER	Delhaize		Certification	Réussite	
7	Carte CL	CL			Authentification	Réussite	
8	Carte CL		Delhaize		Authentification	Echec	Delhaize n'est pas accrédité à utiliser les services de la carte CL

Tableau 7 - 4 : Résultat du test sur un cas concret.

Chapitre 8 : Exemple

Sommaire

Au cours de ce chapitre, nous allons vous présenter l'application en utilisant un cas simple. Cette illustration se base sur des pages écran expliquant à la fois le fonctionnement de l'application et le protocole.

8.1 Schéma du cas concret

Nous allons nous baser sur le cas présenté à la Figure 8 - 1.
Nous pouvons y distinguer trois acteurs :

- un émetteur,
- une carte : C_BBL,
- un prestataire de service : BBL.

Nous identifierons trois phases :

- deux certifications :
  - une entre la carte et l'émetteur,
  - une entre l'émetteur de la carte et le prestataire de services.
- une authentification entre la carte et le prestataire de services.

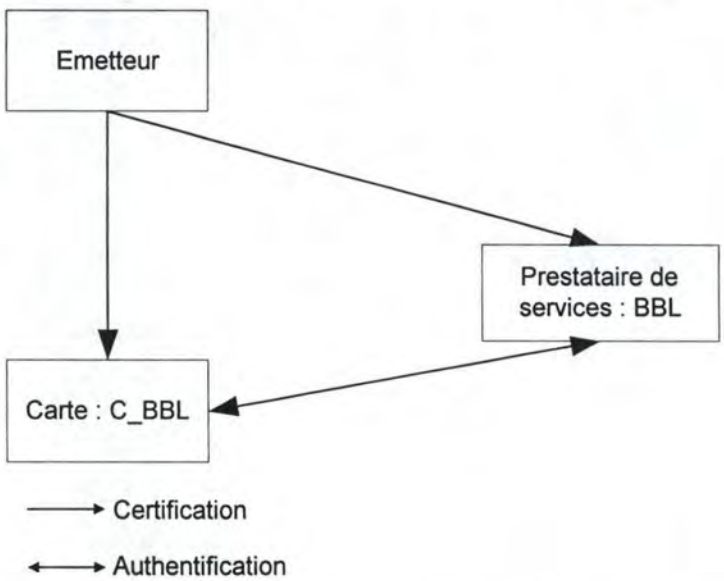


Figure 8 - 1 : Schéma de base.

8.2 Les étapes de la démo

8.2.1 Les créations de partenaires

Avant d'effectuer les authentifications et certifications, l'utilisateur devra créer les différents partenaires : une carte et un prestataire de services (Figure 8 - 2). Pour ce faire, il va identifier les partenaires en leur donnant un nom.

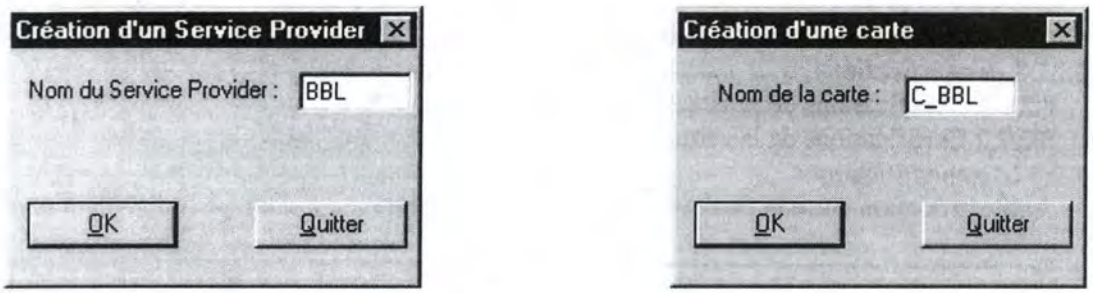


Figure 8 - 2 : Création des partenaires.

8.2.2 Les certifications

Une fois les partenaires créés, l'utilisateur, conformément à la Figure 8 - 1, doit les certifier.

8.2.2.1 Certifications du prestataire de services

L'utilisateur sélectionne dans une liste le prestataire de services auquel il souhaite voir attribuer un certificat (1). Une fois cette opération effectuée, il clique sur le bouton [Suivant] afin de débuter la certification. Les données du certificat s'affichent alors, l'utilisateur clique à nouveau sur [Suivant] afin de transmettre les données à l'émetteur (2). L'émetteur crée alors le certificat (3) qu'il renvoie au prestataire de services (4).

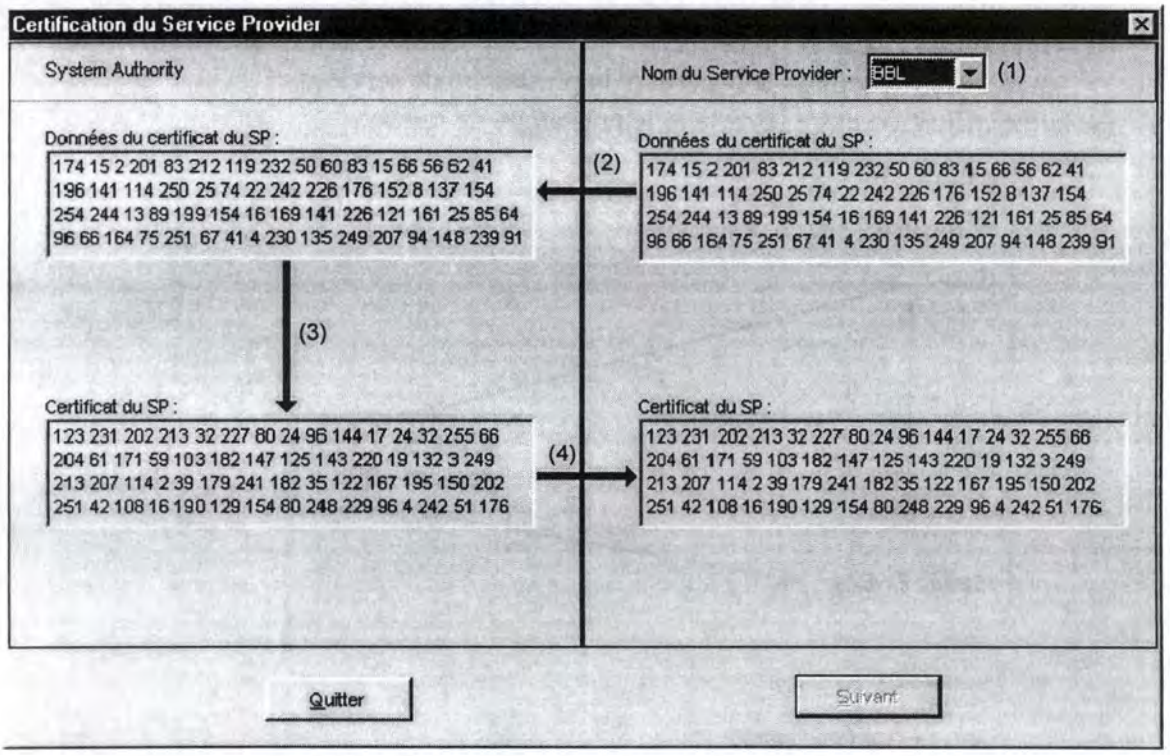


Figure 8 - 3 : Certification du prestataire de services.

8.2.2.2 Certifications de la carte

Cette certification (Figure 8 - 4) de la carte se déroule de la même manière que celle que nous venons de voir pour le prestataire de services.

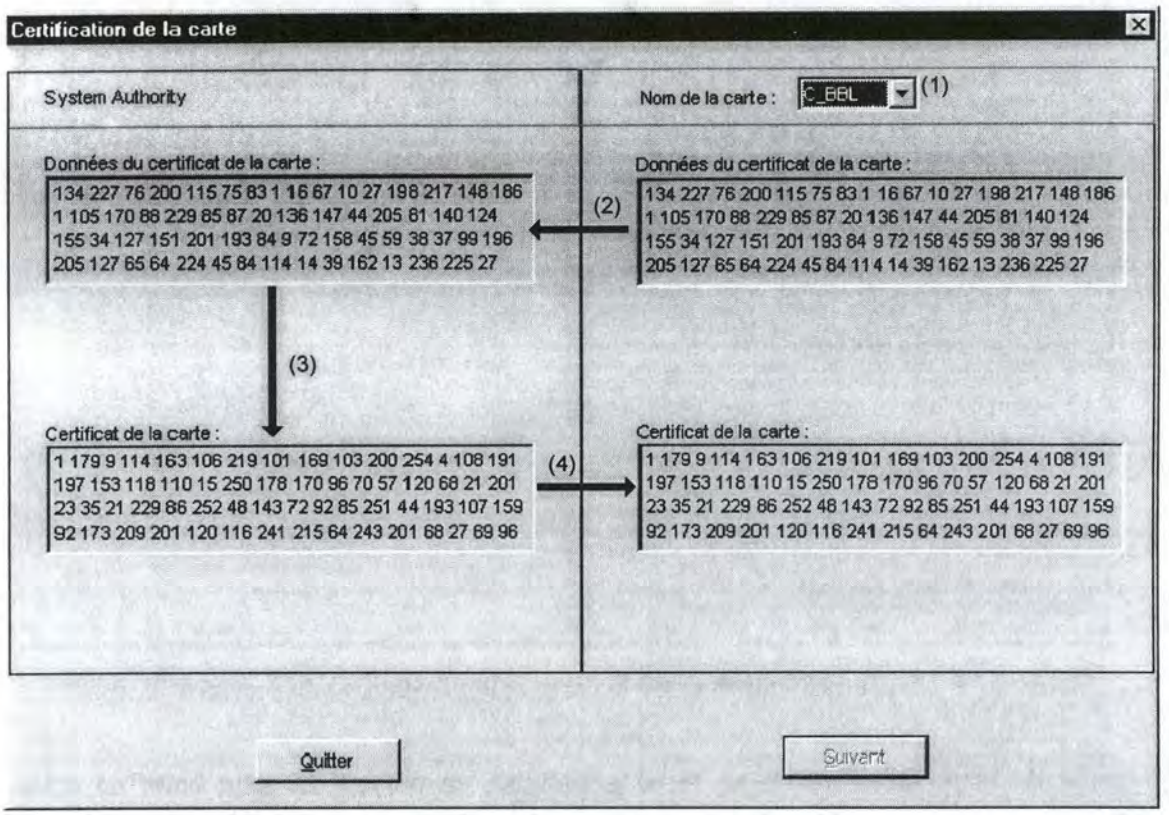


Figure 8 - 4 : Certification de la carte.

8.2.3 L'authentification

Une fois les partenaires certifiés l'authentification peut commencer.

Authentification entre la carte et le prestataire de services

L'utilisateur est invité à choisir les acteurs qu'il désire mettre en présence au cours de l'authentification (Figure 8 - 5). Pour ce faire il choisit le nom du prestataire de services (1) et de la carte (2). Une fois cette opération effectuée, il peut débuter l'authentification en appuyant sur le bouton [Authentification]. Les données du certificat du prestataire de services ainsi que son certificat sont affichés. L'utilisateur clique alors sur le bouton [Suivant] pour transmettre à la carte le certificat et le nombre aléatoire du prestataire de services (3). La carte va alors procéder au déchiffrement de ce certificat (4). Au terme de cette étape, la carte a pu déchiffrer le certificat du prestataire de services. En effet, il s'agit d'un certificat fourni par l'émetteur. L'utilisateur est ensuite invité à cliquer sur le bouton [Suivant] afin de passer à la deuxième étape de l'authentification.

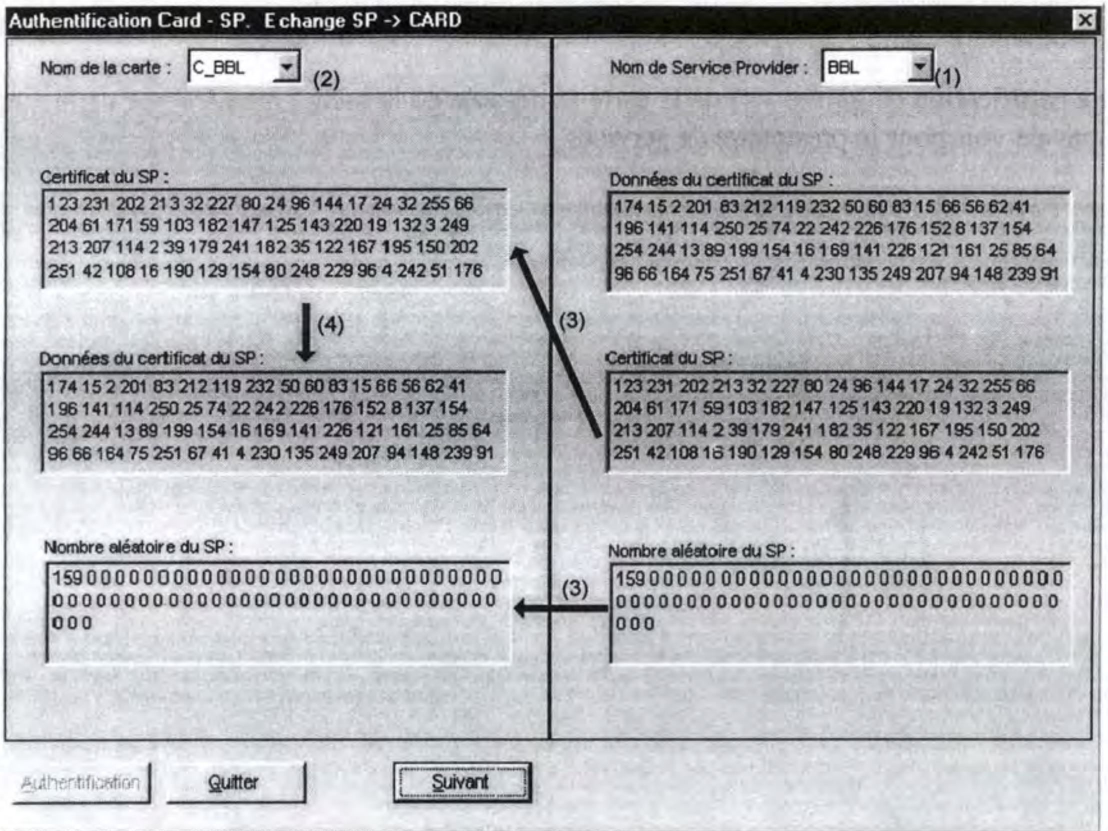


Figure 8 - 5 : Authentification entre la carte et le prestataire de services : étape 1.

Au terme de cette première étape, la carte indique, au moyen de cette boîte de dialogue (Figure 8 - 6) qu'elle a pu vérifier le certificat du prestataire de services.

Figure 8 - 6 : Message d'information à la première étape.

Dans la deuxième étape (Figure 8 - 7), l'utilisateur clique sur le bouton [Suivant] afin de poursuivre la démo. La carte envoie son certificat, le nombre aléatoire chiffré du prestataire de services ainsi que son nombre aléatoire (5). Le prestataire de services va déchiffrer le certificat de la carte (6) et le nombre aléatoire chiffré (7). Au terme de cette étape, le prestataire de services a pu authentifier la carte car le nombre aléatoire envoyé est le même que celui qu'il a reçu. Au terme de cette étape, l'utilisateur est invité à appuyer sur le bouton [Suivant] afin de passer à l'étape suivante.

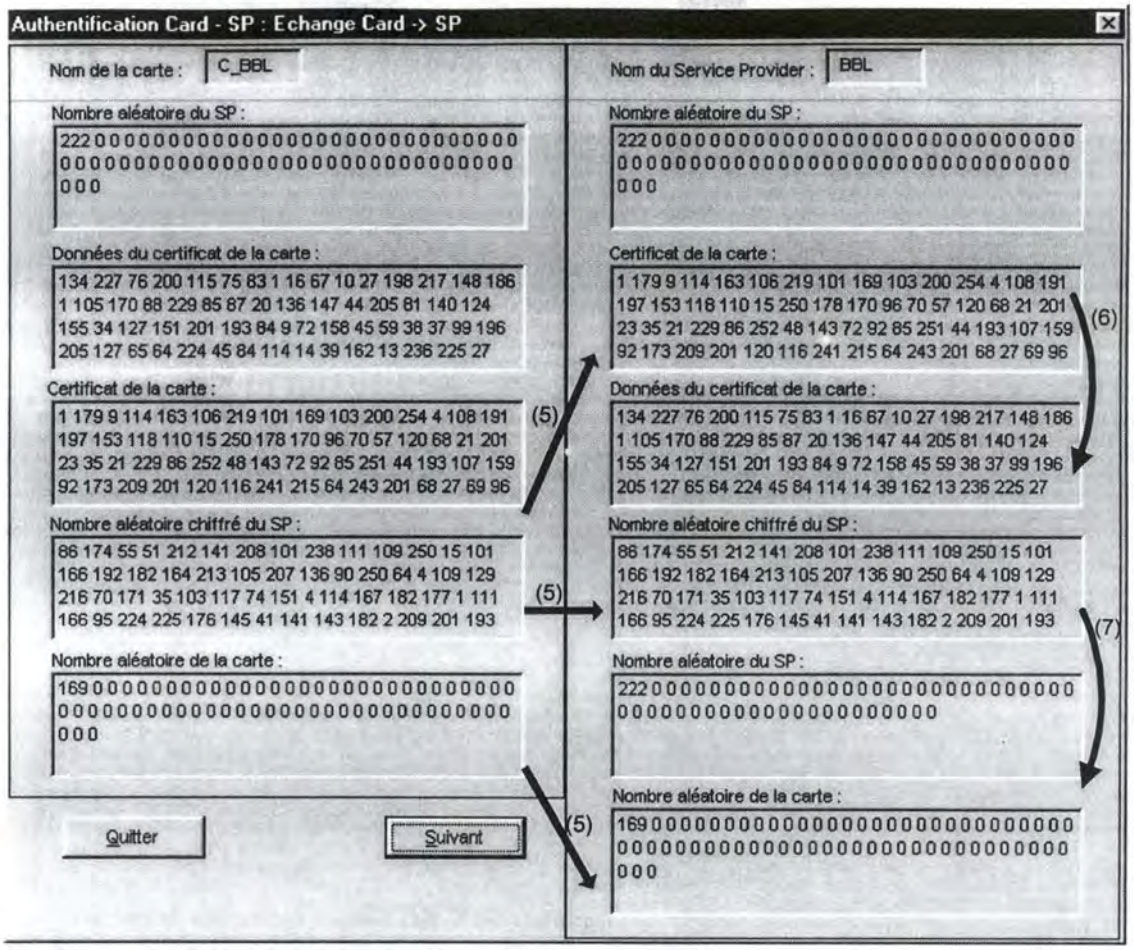


Figure 8 - 7 : Authentification entre la carte et le prestataire de services : étape 2.

Au terme de cette deuxième étape, le prestataire de services indique, au moyen de boîtes de dialogue (Figure 8 - 8) d'une part qu'il a pu vérifier le certificat de la carte et d'autre part qu'il a pu authentifier la carte.

Figure 8 - 8 : Message d'information à la deuxième étape.

Dans cette dernière étape (Figure 8 - 9) le prestataire de services va chiffrer le nombre aléatoire (8) qu'il a reçu de la carte et le lui renvoyer (9). La carte va alors déchiffrer ce nombre et le comparer à celui qu'elle avait envoyé (10). Au terme de cette étape la carte a authentifié le prestataire de services.

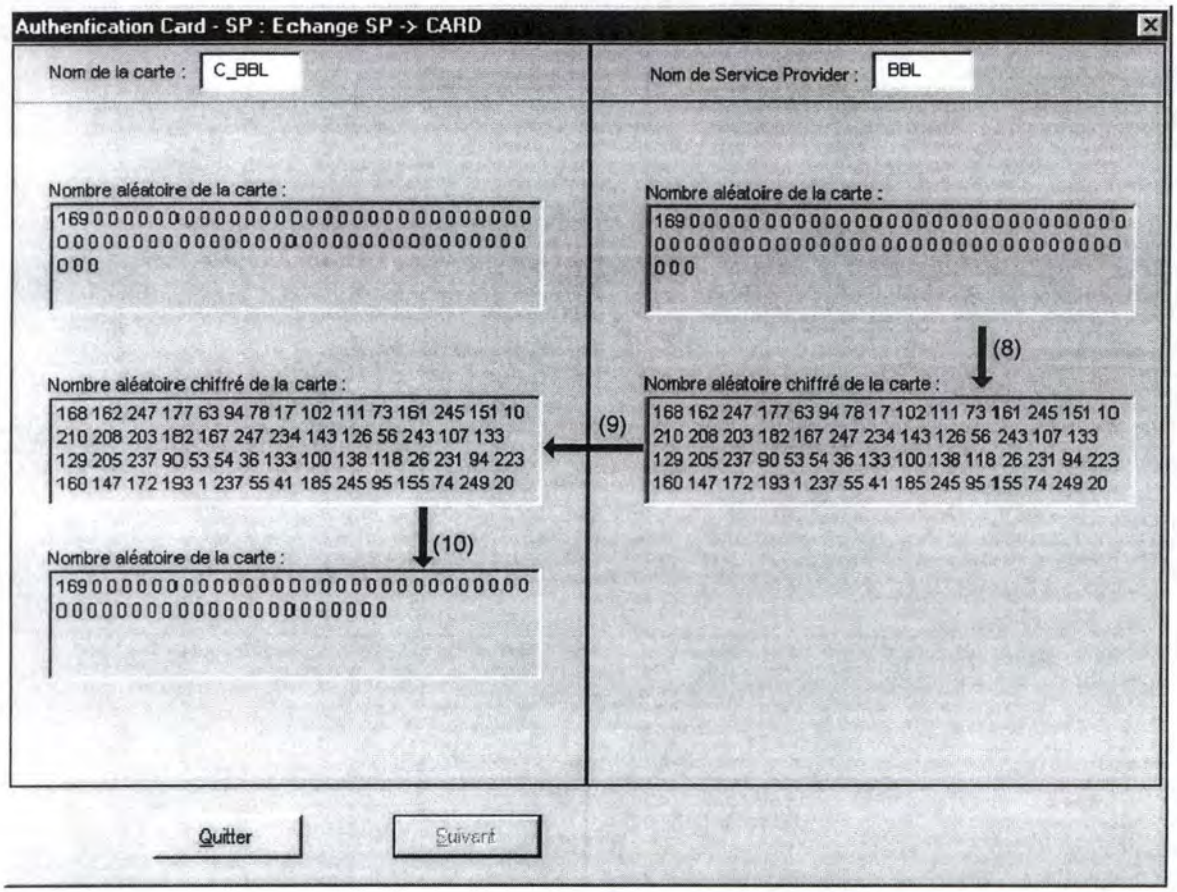


Figure 8 - 9 : Authentification entre la carte et le prestataire de services : étape 3.

Au terme de cette dernière étape, le carte indique, au moyen d'une boîte de dialogue (Figure 8 - 10) qu'elle a pu authentifier le prestataire de services.

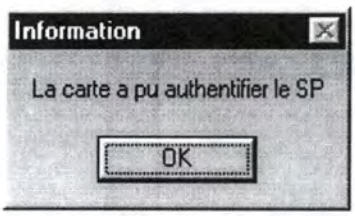


Figure 8 - 10 : Message d'information à la dernière étape.

La carte C_BBL et le prestataire de services BBL se sont mutuellement authentifiés. Le prestataire de services est dès lors autorisé à charger des applications dans la carte.

Conclusion générale

Conclusion générale

Sommaire

Résumé

Au cours de ce mémoire, l'objectif principal était de présenter et d'implémenter le protocole de sécurité des accès des partenaires à la carte générique. Ce protocole s'inscrit dans un projet appelé Combo. Ce projet est développé par l'équipe Recherche de GEMPLUS et a pour but de développer une carte à puce générique.

Nous ne pouvions vous présenter ce protocole sans vous parler d'abord des cartes à puce en général, de la sécurité et par conséquent de cryptographie. Nous avons pu ainsi définir les termes clés permettant de comprendre le protocole.

Les bases étant posées, nous avons pu vous expliquer ce protocole, la manière de l'implémenter et terminer ensuite avec quelques tests qui illustrent la manière dont ce protocole fonctionne.

Intérêts

Beaucoup de nouveaux domaines ont été abordés au cours de ce travail. La carte à puce, la sécurité et la cryptographie sont des matières qui nous étaient totalement inconnues en débutant ce stage.

L'intérêt de ce stage était par conséquent double. D'une part, il nous a permis de mettre en pratique des connaissances acquises lors des quatre années d'étude. Nous pensons aux spécifications, à la programmation en C, à la rédaction d'un rapport. D'autre part, il nous a également permis de découvrir et d'apprendre des matières inconnues telles que la cryptographie, les cartes à puce, la sécurité dans les cartes, ...

Nous pourrions également ajouter qu'il nous a appris à travailler dans une équipe de recherche et à participer à la vie de cette équipe au sein de l'entreprise, et finalement à gérer notre temps afin d'atteindre le but fixé dans les délais convenus.

Perspectives

Néanmoins, ce travail n'est encore que partiel. En effet, de nombreuses perspectives s'offrent à lui.

Les cartes GPK2000 sont maintenant disponibles. Par conséquent, des modifications de l'application pourraient être envisagées afin de la faire fonctionner à l'aide d'une carte à puce.

Nous n'avons considéré dans notre protocole qu'un émetteur. Nous pouvons envisager de compléter ce protocole en permettant l'existence de plusieurs émetteurs.

Nous avons choisi de simuler les cartes, les prestataires de services, les clients et l'autorité de certification à l'aide d'une seule application. Un nouveau défi serait de réaliser une application pour chacun de ces acteurs. Nous aurions alors des applications indépendantes communiquant au travers du protocole.

Bibliographie

Bibliographie

[BKA,93] Bruton S. Kaliski Jr. - An Overview of the PKCS Standards - 1 Novembre 1993.

[BMO,74] *Brevet Moreno* : dépôt du 25 mars 1974 - Institut National de la Propriété Industrielle - numéro de publication 2266222.

[BSH,94] Bruce Schneier - *La cryptographie appliquée : protocoles, algorithmes* - International Thomson Publishing France, 1 rue Saint-Georges, 75009 Paris - 1994.

[ECH,94] L'écho des Recherches n°158 spécial paiement électronique, Centre National d'Etudes des Télécommunications - Ecole Nationale Supérieure des Télécommunications - 4ème Trimestre 1994.

[GEM,97] Gemplus - *Présentation de la Société* - Gemplus 1997.

[GPK,96] Gemplus - Formation interne : Carte à clé publique - Gemplus 1996.

[JJV,96] Jean-Jacques Vandewalle - *Projet Osmose (Operating System and Mobile Services) ; modélisation, implémentation et interopérabilité de services carte à microprocesseur par l'approche orientée objet* - Université des sciences et technologies de Lille, 26 Août 1996.

[JMC,90] John McCrindle - *Smart Cards* - IFS Publications/Springer-Verlag, 1990.

[JRA,97] Jean Ramaekers - *Cours de sécurité* - FUNDP 1997.

[JSV,87] Jemore Svigals - *Smart Cards : The new bank cards* - MacMillan Publishing Company, 1987.

[PB,PG,JJV,96] *How Smart Cards can take benefits from Object-Oriented Technologies* - Cardis 1996.

[PCO,96] Projet Combo, Equipe Recherche de GEMPLUS, 1996.

[RBR,88] Roy Bright - *Smart Cards : Principles, Practice, Applications* - Hellis Horwood Limited, 1988.

[RSA,95] PKCS #11 : Cryptographic Token Interface Standard - RSA Laboratories - 28 Avril 1995.

[SC7,93] *Smart Card News : Smart Card Tutorial Part 7* - Mars 1993.

[SC9,93] *Smart Card News : Smart Card Tutorial Part 9* - Mai 1993.

[RMN,MDS,78] *Using Encryption for Authentication in Large Networks of Computers* - Communications of the ACM, vol. 21, num. 12 - Decembre 1978.

Annexes

Annexes

Sommaire

Les algorithmes présentés ci-dessous sont tirés du livre de Bruce Schneier [BSH,97] et du cours de M. Ramaekers [JRA,97].

L'objectif de ce mémoire n'étant pas d'effectuer une description complète des algorithmes à clé publique et des algorithmes à clé secrète, nous n'entrerons pas dans le détail des algorithmes présentés ci-dessous.

L'algorithme RSA sera néanmoins expliqué avec plus de précisions car il intervient dans le protocole d'accès des partenaires à la carte.

A0. L'algorithme DES

Vu la complexité de l'algorithme DES, notamment en ce qui concerne le calcul des clés intermédiaires et des permutations initiales et finales, nous n'exposerons que les principes de base.

Description de l'algorithme

Le DES (Figure 1) est un algorithme à clé secrète dont le chiffrement et le déchiffrement de messages s'effectuent par blocs. Il s'agit d'un algorithme symétrique c'est-à-dire que l'algorithme de déchiffrement est identique à l'algorithme de chiffrement et que la clé de chiffrement est identique à la clé de déchiffrement.

Il chiffre des données divisées en blocs de 64 bits à l'aide de clés intermédiaires de 56 bits calculées à partir d'une clé secrète. Le résultat du chiffrement est lui aussi d'une longueur de 64 bits. Le chiffrement d'un message est réalisé au terme de 16 itérations comme le montre la Figure 1. Sur cette figure, le « + » représente un ou-exclusif.

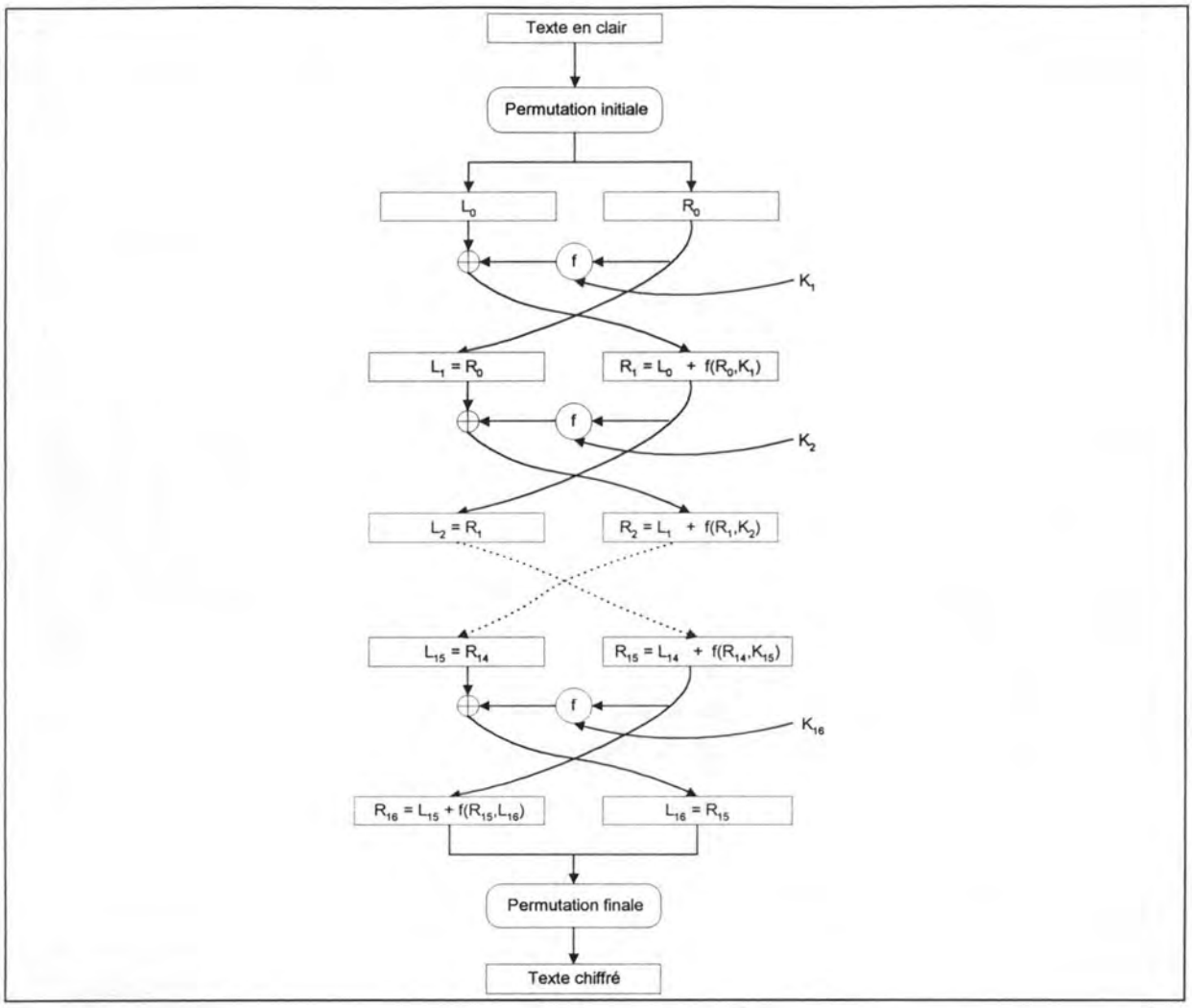


Figure 1 : Algorithme DES.

A1. L'algorithme à clé publique RSA

L'algorithme qui sera présenté au cours de ce chapitre sera l'algorithme RSA (Figure 2) ou protocole RSA, baptisé d'après le nom de ses inventeurs : Ron Rivest, Adi Shamir et Leonard Adleman.

Il s'agit un d'algorithme de chiffrement-déchiffrement de messages et signature-vérification. Ce protocole est à clé publique : chaque client possède une clé publique et une clé privée. La clé publique que tout le monde peut connaître, permet de chiffrer un message; par contre, pour déchiffrer un message, il est nécessaire de posséder la clé privée.

Par conséquent, tout le monde peut chiffrer un message et donc envoyer un message confidentiel, mais seul le propriétaire de la clé privée peut le déchiffrer.

Le protocole

Prenons deux acteurs : Alice et Bob.
Alice veut utiliser le RSA.

Elle génère alors sa clé privée et publique de cette manière :

- Alice choisit deux nombres premiers p et q différents et de grande taille.
- Alice choisit un entier e inversible modulo $\phi(n)$ et le publie.
- Alice calcule l'inverse d de e modulo $\phi(n)$.

La clé publique est donc constituée de n et e. La clé privée est d. Les entiers p, q et $\phi(n)$ doivent rester secrets, car ils permettent de retrouver d.
Pour calculer d, il faut connaître $\phi(n)$ qui vaut :

$$\begin{aligned}\phi(n) &= (p-1)(q-1)\\ &= pq - (p+q) + 1\\ &= n+1 - (p+q)\end{aligned}$$

Si Bob veut envoyer un message m à Alice, il doit calculer $u = m^e \bmod n$ et envoie u à Alice. Bob peut effectuer ce calcul car e et n sont publiques.

Pour déchiffrer le message, Alice calcule $v = u^d \bmod n$. Comme d est l'inverse de e modulo $\phi(n)$, on a $v = m$. Donc le message est déchiffré. Les nombres n et m doivent être premiers entre eux.

En résumé :

Alice choisit deux grands nombres premiers p et q.
Elle calcule e tel que pgcd $(e,(p-1)(q-1)) = 1$.
Elle calcule d tel que $ed = 1 \bmod (p-1)(q-1)$.
La clé publique d'Alice est $K_p = (e,n)$ avec $n = p * q$.
La clé secrète d'Alice est $K_s = d$.

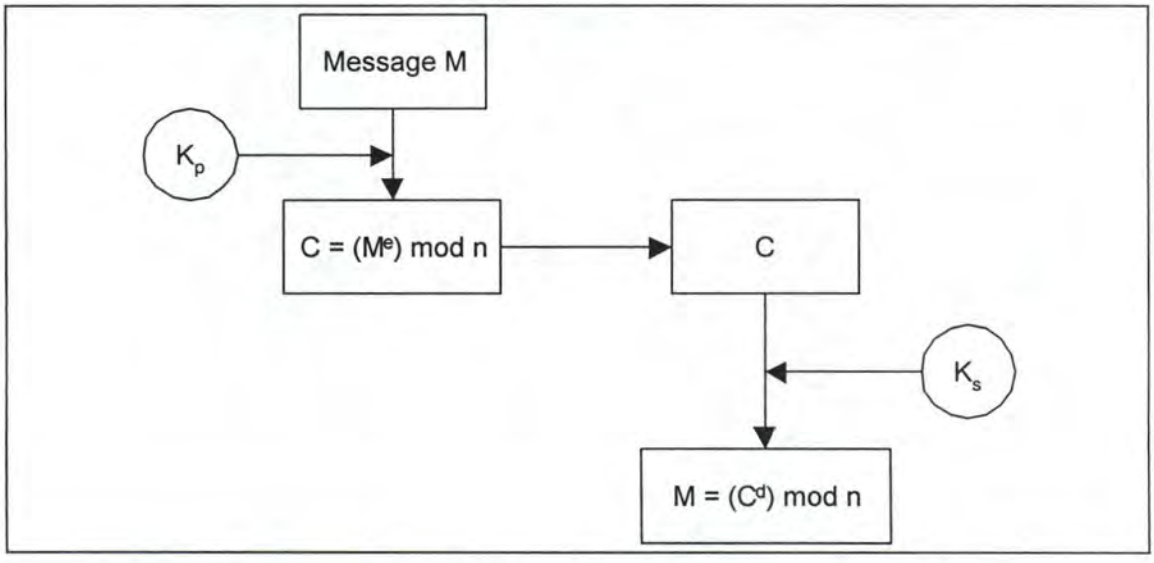


Figure 2 : Algorithme RSA.

La méthode de réalisation : algorithme de calcul des clés

- Pour générer les clés, choisissez deux grands nombres premiers, p et q. Ces deux nombres sont premiers entre eux si le pgcd entre eux deux est égal à 1.

- Calculez le produit n = p * q

- Choisissez une clé de chiffrement aléatoire e tel que e et (p - 1) * (q - 1) soient premiers entre eux. Utilisez l'algorithme d'Euclide pour calculer la clé de déchiffrement d de telle manière que:

  $e * d \equiv 1 \pmod{(p-1) * (q-1)}$

  $d = e^{-1} \bmod ((p-1) * (q-1))$

- Pour chiffrer un message m, découpez celui-ci en blocs numériques tel que chaque bloc ait une représentation unique modulo n. Le message chiffré c sera constitué de manière similaire de blocs c_i de même longueur. La formule de chiffrement est simplement :

  $c_i = (m_i)^e \bmod n$

- Pour déchiffrer un message, prenez chaque bloc c_i et calculez :

  $m_i = (c_i)^d \bmod n$

Le choix de chiffrer avec e ou d est arbitraire. Néanmoins, nous verrons plus tard les raisons qui se cachent derrière ces choix.

Un exemple

Prenons $p = 47$ et $q = 71$.
Alors $n = p * q = 3337$.
La clé de chiffrement *e* ne doit pas avoir de facteur commun avec :

$(p - 1) * (q - 1) = 3220$

Choisissons *e* égal à 79.
Dans ce cas :

$d = 79^{-1} \bmod 3220 = 1019$

Nous pouvons publier *e* et *n* et garder *d* secret.
Soit le message *m* suivant :

m = 688232687966683

Nous divisons ensuite le message en blocs de 3 chiffres :

$m_1 = 688$
$m_2 = 232$
$m_3 = 687$
$m_4 = 966$
$m_5 = 668$
$m_6 = 3$

Le premier bloc est chiffré par :

$688^{79} \bmod 3337 = 1570 = c_1$

En effectuant la même opération sur tous les blocs, nous obtenons :

c = 1570 2756 2091 2276 2423 158

Pour déchiffrer, le message il faudra effectuer :

$1570^{1019} \bmod 3337 = 688 = m_1$

A2. L'algorithme DSA

Le DSA (Digital Signature Algorithm) a été inventé par le National Institute of Standards and Technology. Il s'agit d'un algorithme à clé publique. Il ne réalise que de la signature et de la vérification contrairement au RSA qui réalise également du chiffrement et du déchiffrement.

Description de l'algorithme

L'algorithme utilise les paramètres suivants :
p : un nombre premier de L bits avec L allant de 512 à 1024 et multiple de 64,
q : un nombre premier avec $p-1$ d'une longueur de 160 bits,
$g = h^{(p-1)/q} \bmod p$ avec h un nombre plus petit que $p-1$ tel que $h^{(p-1)/q} \bmod p$ est plus grand que 1,
x : un nombre plus petit que q,
$y = g^x \bmod p$.

L'algorithme utilise une fonction de hachage $H(m)$. Une fonction de hachage est une fonction mathématique qui recevant une chaîne de longueur variable, la convertit en une chaîne de longueur fixe plus petite. Il s'agit d'une fonction à sens unique. Etant donné le résultat du hachage, il est impossible de retrouver la chaîne en entrée.

Les paramètres p, q et g sont publiques. La clé privée est x et la clé publique est y.

Alice souhaite signer un message m. Pour ce faire:

- Alice génère un nombre aléatoire $k < q$
- Alice génère
  $r = (g^k \bmod p) \bmod q$
  $s = (k^{-1}(H(m) + xr)) \bmod q$

Les paramètres r et s sont sa signature. Elle les envoie à Bob.

Bob va vérifier la signature comme suit :
$w = s^{-1} \bmod q$
$u1 = (H(m) * w) \bmod q$
$u2 = (rw) \bmod q$
$v = ((g^{u1} * y^{u2}) \bmod p) \bmod q)$

Si $v = r$ alors la signature est vérifiée.

A3. L'algorithme Diffie-Hellman

Diffie-Hellman fut le premier algorithme à clé publique inventé. Il doit sa sécurité à la difficulté de calculer des logarithmes discrets dans un domaine fini. Il ne peut pas être utilisé pour le chiffrement et le déchiffrement de messages mais est utilisé pour la distribution de clés.

Description de l'algorithme

Alice et Bob conviennent de nombres premiers n et g tel que g est un nombre premier mod n. Les deux nombres n'ont pas besoin d'être gardés secrets.

Le protocole est alors le suivant :

- Alice choisit un grand nombre aléatoire x, calcule $X = g^x \ mod \ n$ et l'envoie à Bob.
- Bob choisit un grand nombre aléatoire y, calcule $Y = g^y \ mod \ n$ et l'envoie à Alice.
- Alice calcule alors $k = Y^x \ mod \ n$.
- Bob calcule alors $k' = X^y \ mod \ n$.

k et k' sont égaux à $g^{xy} \ mod \ n$.

Quelqu'un qui espionnerait les échanges ne pourrait calculer ces valeurs. Il ne connaîtrait que n, g, X et Y. A moins que de savoir calculer des logarithmes discrets et donc de retrouver x et y, il ne pourrait pas résoudre le problème.

Par conséquent, k est la clé secrète qu' Alice et Bob calculent indépendamment.